Leveraging Interactive Human–AI Collaboration Methods to Enhance Key Stages of Programming Workflows

by

Xuye Liu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Xuye Liu 2026

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Tongshuang Wu Assistant Professor, Human-Computer Interaction Institute, Carnegie Mellon University
Supervisor	Jian Zhao Associate Professor, Cheriton School of Computer Science, University of Waterloo
Internal Members	Xi He Assistant Professor, Cheriton School of Computer Science, University of Waterloo
	Edith Law Associate Professor, Cheriton School of Computer Science, University of Waterloo
Internal-external Member	Mark Hancock Professor, Management Science and Engineering, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This dissertation incorporates first-authored and co-first-authored peer-reviewed material published in conference proceedings by the Association for Computational Linguistics (ACL) and the Association for Computing Machinery (ACM). For material published in ACM venues, ACM's policy is as follows[1]:

"Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected)."

The following list serves as a declaration of the works included in this dissertation. The material is expanded and revised from the original publications.

Portions of Chapter 3: Xuye Liu, Dakuo Wang, April Yi Wang, Yufang Hou, and Lingfei Wu. HAConvGNN: Hierarchical Attention Based Convolutional Graph Neural Network for Code Documentation Generation in Jupyter Notebooks. In *Findings of EMNLP 2021*, pp. 4473–4485. https://doi.org/10.18653/v1/2021.findings-emnlp.381. I led the project and drafted the paper. Coauthors provided guidance and revisions.

Portions of Chapter 4: Xuye Liu, Tengfei Ma, Yimu Wang, Fengjie Wang, and Jian Zhao. NBDESCRIB: A Dataset for Text Description Generation from Tables and Code in Jupyter Notebooks with Guidelines. In *Findings of ACL 2025*, pp. 26584–26606. https://doi.org/10.18653/v1/2025.findings-acl.1364. I led the project and drafted the paper. Coauthors helped refine evaluation and provided revisions.

Portions of Chapter 5: Fengjie Wang*, Xuye Liu*, Oujing Liu, Ali Neshati, Tengfei Ma, Min Zhu, and Jian Zhao. (*Equal contribution.) Slide4N: Creating Presentation Slides from Computational Notebooks with Human-AI Collaboration. In *CHI '23*, pp. 1–18. https://doi.org/10.1145/3544548.3580753. I co-led this project with Fengjie Wang. Other coauthors provided feedback and revisions.

Portions of Chapter 6: Xuye Liu, Yuzhe You, Xinrong Qiu, Tengfei Ma, and Jian Zhao. MACEDON: Supporting Programmers with Real-Time Multi-Dimensional Code Evaluation and Optimization. In *UIST '25*. https://doi.org/10.1145/3746059.3747655. I led the project and drafted the paper. Coauthors provided guidance and revisions.

[1] https://authors.acm.org/author-resources/author-rights

Abstract

Beyond writing code, programmers routinely move through several complementary tasks as they develop, refine, and share their work. These workflows typically involve recurring stages: understanding and documenting code, checking correctness and debugging, improving efficiency and scalability, and sharing results with others. Each stage has its own challenges: documentation often becomes outdated or inconsistent with evolving code, debugging can be time-consuming and opaque, performance improvements require balancing competing goals (*e.g.*, speed, memory, and clarity), and communicating results usually demands extra manual effort.

This thesis investigates how human–AI collaboration can support programmers across four key stages of the workflow. To address these challenges, I begin by studying the needs and practices of programmers to understand where current tools fall short. Based on these insights, I design interactive systems that integrate with common tools such as computational notebooks and IDEs and operate on invariant components (code cells, execution outputs, text) so results remain compatible with common practices. Across the four stages, these systems provide context-aware code understanding across multiple cells, purpose-driven documentation from code and its execution results for different communicative purposes, presentation slides from code and results, and real-time, multi-dimensional code evaluation and optimization support during development, with authors remaining in control to inspect, edit, and refine outputs throughout. I conduct user studies and case studies to evaluate system usability and to assess how these approaches improve programmers' productivity, confidence, and ability to share their work.

Acknowledgements

I would like to thank my advisors, Jian Zhao, and close collaborators Tengfei Ma, for their guidance, support, and trust throughout my research path. Their mentorship has been essential at key moments, from my very first experience with research to the completion of this dissertation. I am grateful for the freedom they gave me to explore my ideas while providing timely feedback and direction when needed.

I thank Xi He, Edith Law, Mark Hancock, and Tongshuang Wu for serving on my committee and for carefully reading this thesis. Their questions and comments during the defense made it an insightful and memorable moment, and I appreciate having such a supportive committee.

I thank my friends, Ryan Yen, JianLin Li, Kunzheng Yu, Pengyu Yi, Xinyu Shi, Xueguang Ma, and Yubo Wang for making my PhD experience enjoyable. Sharing coffees, lunches, dinners, and late-night discussions with you has been a precious part of this journey, and I am grateful for your friendship and support.

I thank my collaborators, Pengcheng An, Yimu Wang, and Fengjie Wang, for the many hours we spent together on projects that shaped this dissertation. Your ideas, feedback, and willingness to iterate have been invaluable.

I thank the friends and colleagues in the HCI lab for creating a supportive and welcoming environment. Thank you for the discussions, coffee runs, user studies, and pushes for deadlines that made this work possible.

To my mother, my father and all my other family members, for always believing in me, supporting me, and trusting my choices, even from across the ocean. Your encouragement has been a constant source of strength throughout my thesis and beyond.

Table of Contents

Examining Committee Membership ii

Author's Declaration iii

Statement of Contributions iv

Abstract v

Acknowledgements vi

List of Figures xiv

List of Tables xviii

1 Introduction 1

1.1 Research Objectives and Overview 6

1.2 Contributions 7

1.3 Dissertation Outline 9

2 Related Work 12

2.1 Programming Workflows 12

2.2 Benefits of Human-AI Collaboration in Programming 13

2.3 Existing Tools and Identified Gaps 14

2.3.1 Code Understanding . . . 14
2.3.2 Documentation . . . 14
2.3.3 Communication and Presentation . . . 15
2.3.4 Code Evaluation and Optimization . . . 15

3 Context-Aware Code Understanding across Multiple Code Cells 17
3.1 Motivation . . . 17
3.2 Related Work . . . 21
3.3 *notebookCDG* Dataset . . . 22
3.3.1 Data Preprocessing . . . 22
3.3.2 Dataset Core Statistics . . . 23
3.4 Approach . . . 24
3.4.1 Model Input . . . 24
3.4.2 Embeddings . . . 25
3.4.3 Encoder . . . 25
3.4.4 HAConvGNN . . . 25
3.5 Experimental Setup . . . 26
3.5.1 Implementation Details . . . 26
3.5.2 Baselines . . . 26
3.5.3 Experimental Details . . . 28
3.6 Automated Evaluation . . . 28
3.7 Human Evaluation . . . 31
3.8 Comparison With Transformers . . . 32
3.9 Downstream User Application . . . 33
3.10 Conclusion and Future Work . . . 33
3.11 Ethical Concern . . . 34

4 Generating Documentation from Code and Results for Different Purposes **35**

4.1 Motivation . . . 35
4.2 Related Work . . . 38
4.3 TCDG - Task Description . . . 39
4.3.1 Input . . . 39
4.3.2 Output . . . 40
4.4 NBDESCRIB . . . 40
4.4.1 Data Collection . . . 40
4.4.2 Data Preprocessing . . . 41
4.4.3 Guideline Category Description . . . 42
4.4.4 Train / Dev / Test Splits . . . 44
4.5 Experiments . . . 44
4.5.1 Evaluation Metrics . . . 45
4.5.2 Results . . . 45
4.5.3 Human Evaluation . . . 47
4.6 Error Analysis . . . 47
4.7 Downstream User Application . . . 49
4.8 Conclusion . . . 50

5 Creating Presentation Slides from Code and Results **52**

5.1 Motivation . . . 52
5.2 Related Work . . . 55
5.2.1 Presentation Support for Computational Notebooks . . . 55
5.2.2 Slides and Code Documentation Generation . . . 55
5.2.3 AI-Infused Systems for Data Science . . . 56
5.3 Slide4N Design . . . 57
5.3.1 Design Goals . . . 57

5.3.2 System Overview . . . 59
5.3.3 Usage Scenario . . . 59
5.4 Slide4N . . . 61
5.4.1 Locate: Finding Relevant Cells . . . 61
5.4.2 Distill: Generating Contents from Cells . . . 62
5.4.3 Arrange: Organizing and Laying out Contents . . . 64
5.4.4 Refine: Customizing Generated Slides . . . 66
5.5 Slide4N User Interface . . . 68
5.5.1 *Code* . . . 68
5.5.2 *Control* . . . 69
5.5.3 *Slide* . . . 69
5.5.4 *Content* . . . 69
5.5.5 Interactive Linking . . . 70
5.6 User Study . . . 70
5.6.1 Study Design and Data . . . 70
5.6.2 Participants . . . 71
5.6.3 Procedure and Task . . . 72
5.7 Results . . . 73
5.7.1 Questionnaire Responses . . . 73
5.7.2 Slides Quality Evaluation . . . 75
5.7.3 Qualitative Feedback . . . 76
5.8 Discussion . . . 81
5.8.1 Design Implications . . . 81
5.8.2 Limitations and Future Work . . . 84
5.9 Conclusion . . . 86

6 Human–AI Collaboration for Multi-Dimensional Code Evaluation and Optimization **87**

6.1 Motivation . . . 87
6.2 Related Work . . . 90
6.2.1 Generating and Optimizing Code with LLMs . . . 90
6.2.2 Evaluation of LLM-based Code Assistants . . . 91
6.2.3 Positioning Relative to Traditional and LLM-based Tools . . . 92
6.3 Design Process & Goals . . . 92
6.3.1 Interview Process . . . 92
6.3.2 Interview Results . . . 93
6.3.3 Design Guidelines . . . 95
6.4 Define Code Evaluation Method . . . 96
6.4.1 Data Collection . . . 96
6.4.2 Data Analysis and Results . . . 97
6.5 MACEDON: Design and Implementation . . . 99
6.5.1 System Architecture . . . 100
6.5.2 User Interface Design . . . 101
6.5.3 Approach for Multi-Dimensional Code Evaluation and Optimization 101
6.6 User Evaluation . . . 105
6.6.1 Participants . . . 105
6.6.2 Study Protocol . . . 105
6.6.3 Data Collection and Measurement . . . 106
6.6.4 Results . . . 108
6.7 Case Studies . . . 113
6.7.1 Early-Stage Programming Task . . . 114
6.7.2 Programming Evaluation During the Interview . . . 115
6.8 Discussion . . . 117
6.8.1 Automated Code Review Practices in Software Engineering . . . 117

6.8.2 Code Optimization in Programming Differs from Traditional Software Refactoring . . . 118
6.8.3 Limitations and Future Work . . . 119
6.9 Conclusion . . . 120

7 Conclusion **122**
7.1 Summary of Contributions . . . 122
7.1.1 Artifact Contributions . . . 122
7.1.2 Empirical Research Contributions . . . 123
7.2 Supporting Programming Workflows: Authors, Tools, or Human–AI Collaboration? . . . 124
7.3 Integration and Coherence . . . 125
7.4 User Requirements and Evolving AI Capabilities . . . 126
7.4.1 User Populations Across Stages . . . 126
7.4.2 Pipeline Robustness Under Advancing AI Capabilities . . . 127
7.4.3 What Remains Human and What Becomes Automated . . . 127
7.5 Broader and Longer-Term Impact . . . 129
7.6 Limitations and Future Work . . . 130
7.6.1 Validation Gaps Across the Four Projects . . . 130
7.6.2 Proxy Metrics and Evaluation Complexity . . . 131
7.6.3 When AI Assistance May Be Unhelpful . . . 132
7.6.4 Generalizability . . . 133

References **135**

APPENDICES **161**

A HACONVGNN CODE-DOCUMENTATION EXAMPLES **162**
A.1 Code Snippets-Documentation Pair Examples . . . 162

B NBDESCRIB HUMAN EVALUATION DETAILS **167**

B.1 Human Evaluation 167

B.2 Guideline-Code Snippets-Table-Documentation Pair Examples 170

B.3 Kaggle Competition Link 176

B.4 Detail of Baseline Models 176

B.5 Guideline Categories 177

B.6 Prompt for Chain of Thought on GPT-4o 178

B.7 Prompt for In-Context Learning on GPT-4o 179

B.8 G-Eval in Coherence, Consistency, Correctness, and Fluency 180

B.8.1 Definition 180

B.8.2 Prompt Details for obtaining metrics 181

B.9 Table Preprocess Example 184

B.9.1 An example of Original Table Crawled from Kaggle Notebooks . . . 184

B.9.2 Table preprocessing from the the original table in H.1 186

B.9.3 Table Curation from the the table in H.2 187

List of Figures

1.1 Pipeline of complementary supports across programming workflows. Prior work argues that programming is a multi-stage process involving distinct tasks and that programmers work with multiple representations [90, 96, 138, 167, 221]. Top row (tasks): completion [30, 213], annotation [167], synthesis [198, 221], scoring [18, 129]. Middle row (stages): understanding [30, 96], documentation [167], presentation [198, 221], evaluation [18, 129]. Bottom row (representations): code snippets [90, 96], comments [167], artifacts [198, 221], metrics [18, 129]. 3

1.2 Chapters with main research questions (italic), main results, and research path connecting each chapter. Arrows show the dependencies and motivation that lead to the next research question. 11

3.1 HAConvGNN model architecture 24

3.2 Attention visualization for the data point illustrated in Table 3.1. Each row represents a code cell, and each column is a code token. In this example, it shows the second and third token in the second code cell (“ nn_model”, “X”) contribute the most to the predicted documentation in Table 3.1. . . . 28

3.3 Average rated scores given by human evaluators to each method across three dimensions. 29

3.4 We implement a downstream application as a Jupyter Notebook plugin (A) to assist users documentation writing, incorporating the HAConvGNN-predicted results (B) next to an IR-based approach (C), and a user-prompt approach (D). 30

4.1 An example in our proposed NBDESCRIB dataset, which targets generating purpose-driven descriptions based on the input of codes, tables, and user-written guidelines. Previous methods and benchmarks focus on understanding the codes or tables only, which can lead to descriptions that misalign with user intent. 36

4.2 We implement a downstream application as a Jupyter Notebook plugin (A) to assist users in documentation writing for selected rows and columns from different guideline categories (B) and a user-prompt approach (C). 48

5.1 Slide4N is an interactive AI assistant for data scientists to create slides from computational notebooks, which can be used side-by-side with JupyterLab (A). The front-end user interface of Slide4N consists of four interactively linked components: a *Code* (B) that visually summarizes all the cells in a notebook and their relationships, a *Control* (C) that allows users to provide high-level guidance to the slide generation, a *Content* (D) that displays the outline of the working slide deck, and a *Slide* (E) that supports interactive customization and refinement of the generated slides. 60

5.2 The four-step process to arrange contents on a slide (*e.g.*, bullet points, plots, and tables): (1) separate user-selected cells into groups based on the relevance scores; (2) arrange bullet points and cell outputs based on the logical order of cells; (3) align text and plots by linking them (red lines) and construct the backbones within and between groups; (4) map the two-layer backbones to the layout design patterns (see Figure 5.3 for details). 65

5.3 The mapping between slide narrative backbones (in a tree structure) to Bach *et al.*'s layout patterns [11]. G denotes a cell group, B denotes all the bulleted points in a group, M denotes a cell output such as a plot or table, and the gray circle denotes the root node of the tree. 67

5.4 Participants' ratings on the post-study questionnaires including the created slides (Q1-4), the back-end (Q5-12), and the user interface (Q13-21), on a 7-point Likert Scale (1 = "strongly disagree" and 7 = "strongly agree"). . . 73

5.5 Audience ratings on five aspects (R1-5) of the slides created by Slide4N: (a) boxplots of ratings for the 12 decks of slides created using the experimental notebook in study Part 1, and (b) boxplots of ratings for the 6 decks of slides created using participants' own notebooks. The five aspects include overall satisfaction (R1), clarity of structure (R2), ease of understanding the content (R3), proper organization of the content (R4), and aesthetics (R5), on a 7-point Likert Scale (1 = "strongly disagree" and 7 = "strongly agree"). 75

5.6 (a) When the notebook is too long, the rectangles in *Code* can be small and difficult to select. (b)(c) Slide4N sometimes generates less satisfying bullet points and the participants needed to improve the text (indicated by the red boxes) on their slides (left: the original notebook; top-right: AI-generated bullet points; bottom-right: edited bullet points). (d) Slide4N sometimes produces undesired layouts and the participant wanted to make the figures larger. 78

6.1 Distributions of key metrics: clarity, redundancy, time efficiency, space efficiency, and documentation in PIE dataset. 95

6.2 MACEDON is an extension of the Visual Studio Code. It provides three basic dashboards: (A) Evaluation Dashboard provides the current score of their code in five dimensions(clarity, documentation, redundancy, time efficiency, and space efficiency); (B) a button for user to trigger Optimization Dashboard; (C) Optimization Dashboard provides a prioritized list of actionable suggestions based on the real-time evaluation results in each dimension. . . 100

6.3 Results of the post-task questionnaire. The numbers indicate the counts of participants on corresponding scales. 109

6.4 Optimized version of a student's sorting algorithm with improvements guided by MACEDON. Detailed optimization by dimension: (1) **Clarity**: Variable names such as `arr` and `tmp1` were replaced with more descriptive names like `numbers` and `number`, improving readability. The loop to print sorted values was changed to a range-based for loop for more concise and clearer iteration. (2) **Time Efficiency**: The original nested loop sorting algorithm with time complexity $O(n^2)$ was replaced by the C++ standard library function `std::sort`, which uses an optimized quicksort algorithm, improving the sorting time to O(n log n). (3) **Space Efficiency**: Unnecessary temporary variables (e.g., `tmp1`) and redundant conditional checks (e.g., `arr[i] == arr[j]`) were removed. The space overhead was reduced as no extra memory was used for the sorting process. (4) **Documentation**: Added comments explaining the function's purpose, the choice of sorting algorithm, and the logic of the iteration and output steps. These comments make the code easier to understand for beginners and future developers. (5) **Redundancy**: A redundant conditional branch `else if (arr[i] == arr[j])` was removed because it had no functional effect, simplifying the logic and reducing unnecessary operations. 121

List of Tables

3.1 An example of multiple code cells after one documentation block 19

3.2 ***notebookCDG*** dataset statistics. The overall code-to-markdown ratio is 2.2195, which suggests one markdown corresponds to more than one code cells. 21

3.3 ROUGE scores for the baselines, our model, and the ablation models. Results show that our model has higher scores for all three metrics, demonstrating a robust advantage over the code2seq and graph2seq models. 27

3.4 Human Evaluation Result 30

4.1 ***NBDESCRIB*** dataset statistics. 41

4.2 We identify 15 guideline categories based on the types of descriptions in the Markdown cells which are below the code whose output is a table. 43

4.3 ROUGE scores (R-1, R-2, R-L), BERTScore (BERTSc), Pyramid Evaluation (Pyr), and G-EVAL dimensions for the baselines, our model, and the ablation studies. Results show that this task is challenging though we use state-of-the-art text generation models. 45

4.4 An example of code-table cells and different models. Note that the table above is the original table, which will be processed by the table curation in Section 4.4.2 when generating the description. In this case, the curated table only keeps the passenger and age columns. 51

5.1 The association between distinct topics (tags) and slide titles that Slide4N uses to generate title candidates which are further selected using SequenceMatcher. 63

6.1 Evaluation methods, formulas, and optimization examples for each dimension. Notation: $perc(\cdot)$ returns a training-set percentile rank in $[0, 1]$; T_{exec} is execution time; M_{usage} is peak memory usage; $Redundant_{lines}$ counts duplicate/dead lines. For clarity, f_i are style factors with weights w_i ($\sum_i w_i = 1$). For documentation, S_{sim} is the cosine similarity between a code segment and its accompanying natural-language documentation (e.g., CodeBERT embeddings). The ceiling operator $\lceil \cdot \rceil$ maps the $0-1$ value onto the common $1-10$ rubric (top 10% receive 10). 98

6.2 Performance data in two conditions (M: mean, SD: standard deviation): the task completion time (secs), participants' satisfaction with the final code artifact (from -2 to 2), code quality, number of line, and number of tokens. In particular, participants spent less time to complete the task in the MACEDON condition than the Copilot condition (p = .001); participants were more satisfied with the final Code in the MACEDON condition than the Copilot condition (p = .001). 107

A.1 Example: Feature Scaling 162

A.2 Example: Handle Missing Values 163

A.3 Example: Data Augmentation 164

A.4 Example: Plot Model Performance 165

A.5 Example: Count Monthly Mean 166

B.1 Human Evaluation Result. 168

B.2 An example of code and its table output pair(cabin is the Cabin number). In this case, the curated table only keeps the cabin columns. 171

B.3 An example of code and its table output pair (Pclass is the ticket level. The smaller the value, the higher the class). In this case, the table is not curated because ground truth have class and survived variables. 172

B.4 An example of code and its table output pair. In this case, the curated table only keeps the passenger, SibSp, Parch, and FamilySize column. 173

B.5 An example of code and its table output pair. In this case, the curated table only keeps the Accuracy and Model columns. 174

B.6 An example of code and its table output pair. In this case, the curated table only keeps the Wine and pH column. 175

B.7 We identify 15 guideline categories based on the types of descriptions in the Markdown cells below the code whose output is a table. 188

Chapter 1

Introduction

Modern programming increasingly relies on interactive notebook-style environments or full-featured integrated development environments (IDEs), where code authoring, execution, and analysis are tightly coupled [90, 95, 151, 152]. Across contemporary practice, programmers routinely move among data analysis, software development and maintenance, algorithm design, and application prototyping/deployment [20, 90, 96, 138]. Representative professional tools include Jupyter Notebook, Visual Studio Code and IntelliJ IDEA; these tools provide syntax-aware editing, debugging, profiling, and version-control support as core development capabilities [18, 213]. These environments have lowered the barrier to exploratory programming, but they also make programming workflows more multi-faceted. Beyond writing correct code, programmers must understand and document their work, check correctness and debug, improve efficiency while balancing competing goals, and share results with others. Each of these activities poses distinct challenges that shape how programmers interact with tools and artifacts.

Programming workflows are therefore not a single task but a recurring, multi-stage process [96, 138]. Software engineering pipelines have long organized development into distinct phases. Classical models describe flows from analysis through design, implementation, testing, and maintenance [163]; iterative models repeat analysis, design, implementation, and evaluation within each cycle [102]; agile methods compress these phases into short iterations but retain the same core activities [17]. Empirical studies confirm that comprehension, documentation, and testing account for the majority of development effort [103, 212], and that documentation remains important yet frequently neglected [49]. Studies of program comprehension further show that programmers build mental models through hypothesis formation [23], construct multiple representations of code behavior [109, 150], and spend substantial effort seeking and relating information across reading, writing, and

reasoning [97, 170, 173]. In data science contexts, communicating results to stakeholders is a distinct and time-consuming activity that goes beyond documentation [101, 221].

This dissertation identifies four stages that recur across these pipelines: *understanding*, *documentation*, *presentation*, and *evaluation and optimization*. Each stage produces a different primary artifact and serves a different purpose. *Understanding* builds mental models of program behavior through reading, reasoning, and hypothesis testing. *Documentation* records design decisions and rationale to support collaboration and maintenance. *Presentation* communicates results in accessible forms such as slides or reports. *Evaluation and optimization* measures correctness, performance, and readability to guide code improvement. These differences in artifact type and purpose make it useful to support the four stages separately, even though programmers move among them freely rather than following a fixed sequence. Figure 1.1 illustrates this pipeline in three aligned rows, based on literature and the formative studies in Chapters 3–6. The *top row* shows the **tasks** at each stage: *comprehension* (reading, reasoning about, and building mental models of code, supported by tools such as code generation that help verify understanding) [30, 97, 170, 213], *annotation* (documentation writing and maintenance) [167], *synthesis* (slide and report creation for stakeholders) [198, 221], and *scoring* (quality assessment and optimization) [18, 129]. The *middle row* gives the four **workflow stages**: programmers *understand* code by reading, reasoning about structure and behavior, and building mental models [23, 96, 109, 150]—code completion and generation tools assist this process by letting programmers verify hypotheses about what the code should do [30, 213], *document* code to preserve design intent and readability [167, 189], *present* outcomes into reports, slides, dashboards, or shared notebooks [198, 221], and *evaluate* and *optimize* code artifacts across quality dimensions while incorporating peer feedback and release-readiness checks [16, 18, 129]. The *bottom row* lists the **representations** programmers manipulate at each stage: *code snippets* [90, 96], *comments* [167], *artifacts* (slides, reports, dashboards) [198, 221], and *metrics* (quality scores, runtime, memory) [18, 129]; these representations, including code, documentation, and execution outputs, evolve as work progresses [90, 167, 221]. This structure reflects how prior studies describe programming as involving multiple tasks, stages, and artifacts [90, 96, 138, 167, 221].

Although Figure 1.1 presents the four stages from left to right, programmers do not follow a fixed sequence in practice. They can enter any stage at any time and switch freely among stages as their needs change. For example, a programmer may start by evaluating code quality, then return to understanding when the evaluation reveals an unfamiliar module, document the module after gaining understanding, and later present the findings to a colleague. The ordering in the figure reflects a common progression observed in our formative studies and in the literature [96, 138], but it is not prescriptive. Each stage is in-



Figure 1.1: Pipeline of complementary supports across programming workflows. Prior work argues that programming is a multi-stage process involving distinct tasks and that programmers work with multiple representations [90, 96, 138, 167, 221]. Top row (tasks): completion [30, 213], annotation [167], synthesis [198, 221], scoring [18, 129]. Middle row (stages): understanding [30, 96], documentation [167], presentation [198, 221], evaluation [18, 129]. Bottom row (representations): code snippets [90, 96], comments [167], artifacts [198, 221], metrics [18, 129].

dependently accessible, and the tools developed in this dissertation support this flexibility: they operate within existing environments and do not require completion of a prior stage before use.

Each stage of the pipeline poses distinct challenges. During *code understanding*, programmers must read, navigate, and reason about code to build accurate mental models of program behavior [23, 150, 173]. In notebook environments, this challenge is amplified because related code cells span multiple execution steps and interleave with markdown and outputs [90, 167]. Code-completion and generation tools can support this process by helping programmers test hypotheses about what the code should do [30], but understanding itself centers on reading and reasoning rather than on generation. During *documentation*, however, support is far thinner: notebook documentation is frequently outdated or inconsistent with evolving code, because writing and maintaining prose explanations remains a largely manual endeavor [90, 167]. During *presentation*, communicating results to non-technical stakeholders demands extra manual effort: creating slides from their code projects is laborious, repetitive, and error-prone, and generating faithful textual descriptions of tabular outputs involves selecting salient facts without introducing unsupported claims [101, 221]. During *evaluation and optimization*, programmers must juggle correctness, runtime, memory, readability, and redundancy, yet most existing tools surface only one dimension at a time and require separate invocations, making it difficult to see how improvements in one area affect others.

AI-assisted programming tools now support parts of this pipeline, especially code completion. Many ideas have been explored to make such support more capable and more

accessible, but assistance is still concentrated in early-stage coding activities [30, 185, 213]. Later stages such as documentation, presentation, and code quality evaluation and optimization remain comparatively under-served. Beyond code authoring, programmers are expected to write clear documentation for collaborators [167, 189], publish understandable project artifacts for diverse audiences on platforms such as GitHub [38], prepare documents or slides for collaboration and presentations [198, 230], summarize results and share progress with colleagues [101, 221], and satisfy code-quality checks and peer-review requirements before integration or release [16, 129]. In practice, these tasks often require switching across different tools outside notebooks or IDEs, which fragments workflows and introduces inconsistency. Few programmers can claim to execute all of these requirements well in everyday practice, because they frequently clash with existing coding workflows and delivery expectations. Even when experienced programmers can handle these demands, they still patch together workarounds, losing time each time they move from one stage to the next; for novice programmers, the challenge is greater because limited experience often leaves them unsure where to begin and how to proceed [48, 122, 185].

The central task this dissertation supports is helping programmers complete the four key stages of their workflow—understanding, documentation, presentation, and evaluation and optimization—within tools they already use. The goal is not to replace programmers but to reduce the manual effort and brittleness that arise when they move across these stages. Programming involves judgment calls that depend on intent, audience, and context. AI systems can produce drafts, compute metrics, and surface patterns, but they cannot determine whether a documentation paragraph matches the author's intent, whether a slide deck conveys the right message to a particular audience, or whether a performance gain is worth a loss in readability. These decisions require human judgment. This dissertation therefore positions the programmer as the decision-maker and the AI as an assistant that generates starting points for human review.

Fully automated support often falls short because intent, context, and audience needs are difficult to capture without human input; interactive collaboration addresses this by placing the programmer in the loop to verify, steer, and refine AI proposals. To address these issues, this dissertation advances **interactive Human–AI collaboration** as the central approach across all projects, a high-level design paradigm aligned with the thesis title. This approach works by having AI propose drafts, summaries, or feedback, while the programmer inspects, edits, and decides. This keeps outputs grounded in human intent and makes the systems compatible with existing programming tools.

All four projects share a common collaboration pattern that we call the **Propose–Inspect–Refine** loop. In the *propose* step, the AI system generates an initial artifact from code context: a documentation draft, a description, slide content, or quality scores

and suggestions. In the *inspect* step, the programmer reviews the AI output within a familiar tool (a Jupyter plugin or an IDE extension) and judges its correctness, relevance, and completeness. In the *refine* step, the programmer edits, accepts, or rejects the output, and may trigger another round of AI generation. This three-step loop preserves human agency [69]: the programmer remains the decision-maker, and the AI serves as an assistant whose output is always editable. The loop adapts to each stage by varying what is proposed and how inspection occurs:

- **Understanding** (HAConvGNN): the AI proposes context-aware documentation for notebook cells; the programmer inspects the generated text in a Jupyter sidebar and validates or corrects it to deepen comprehension.
- **Documentation** (NBDESCRIB): the AI proposes purpose-driven descriptions grounded in code and tabular output; the programmer reviews factual grounding and revises wording for the target audience.
- **Presentation** (Slide4N): the AI proposes slide content and layout from notebook cells; the programmer previews slides, adjusts scope and structure, and edits individual elements.
- **Evaluation and optimization** (MACEDON): the AI proposes multi-dimensional quality scores and improvement suggestions; the programmer inspects trade-offs across dimensions and decides which optimizations to apply.

Despite the variation in artifacts and tools, every project follows the same principle: the AI generates a starting point, and the programmer steers the result. This shared mechanism makes the collaboration pattern coherent across all four stages.

Rather than pursuing full automation, for each stage of the pipeline this dissertation employs interactive Human–AI collaboration to improve performance. Each project develops an interactive Human–AI system; HAConvGNN and NBDESCRIB additionally conduct analysis and dataset construction before building their systems, grounding the design in data-driven insights. The four systems form a progression from understanding to documentation, to presentation, and finally to evaluation and optimization, with each stage building on the artifacts and constraints of the previous one. This dissertation focuses on these four stages with interactive Human–AI collaboration: 1) **HAConvGNN** targets the understanding stage through context-aware documentation generation in multi-cell notebooks. We construct the notebookCDG dataset and train the HAConvGNN model to better understand code structure and cross-cell dependencies, then design an interactive

Human–AI system (a Jupyter plugin) that helps users obtain conceptual explanations of what the code does, refine and validate AI-generated interpretations, and ultimately understand the code more effectively. Pure automation is brittle because dependencies span cells and meanings depend on context; the collaboration creates a feedback loop where human judgment corrects and steers machine summaries. 2) **NBDESCRIB** focuses on documentation, a critical bridge between code and collaborators. It employs dataset construction (the NBDESCRIB benchmark with purpose-driven guidelines corresponding to communicative purposes such as technical reporting, stakeholder updates, and decision support), model benchmarking and analysis, and an interactive system for collaborative writing. Fully automated text often misses intent or introduces unsupported facts when code outputs are involved; the dataset, guidelines, and interactive tool let users ground descriptions in evidence and revise wording for their audience. 3) **Slide4N** moves from documentation to presentation by helping programmers turn notebooks into presentation artifacts, thereby promoting communication among collaborators. The system integrates model-based slide generation into a JupyterLab plugin that keeps users in control of scope, structure, and edits, reducing manual assembly work while preserving author control. 4) **MACEDON** completes support across the pipeline with evaluation and optimization. We define measurable dimensions (clarity, redundancy, documentation, runtime, memory) and build an IDE extension that surfaces real-time, multi-dimensional feedback and invites iterative human decisions about which improvements matter in a given context. Code quality involves trade-offs that cannot be resolved by a single metric; human–AI collaboration lets programmers inspect evidence and choose what to optimize.

In essence, my research aims to build an integrated programming support stack spanning understanding, documentation, presentation, and evaluation and optimization, all under the unifying lens of **interactive Human–AI collaboration**. Each project develops an interactive Human–AI system where the system drafts and surfaces signals while programmers steer, verify, and refine the results; for HAConvGNN and NBDESCRIB, preparatory work (dataset construction, model training, or analysis) grounds the design before the interactive tool is built. The remainder of this chapter states the research objectives, summarizes the contributions, and outlines the structure of the dissertation.

1.1 Research Objectives and Overview

Simply put, the high-level research objective of this dissertation is to *reduce the manual effort and brittleness (e.g., unreliable or context-insensitive automation) that arise when programmers understand code, document code, present code, and evaluate and optimize*

quality across the programming workflow, while keeping authors in control within their existing tools.

The central approach, as reflected in the thesis title, is **leveraging interactive Human–AI collaboration methods**. My thesis is that:

> *Interactive human–AI collaboration can enhance key stages of programming workflows by supporting programmers in understanding code, documenting code, presenting code and results, and evaluating and optimizing quality, within existing tools and practices and with authors remaining in control.*

This approach is realized through dataset construction, model training, and interactive tool design in varying combinations across the four projects, as summarized above.

To reach this goal, we investigate a set of primary research questions. These questions are related and the output of a project often drove the research questions and design decisions of other projects (Figure 1.2). Each question maps to a chapter: RQ(a) to Chapter 3 (HAConvGNN), RQ(b) to Chapter 4 (NBDESCRIB), RQ(c) to Chapter 5 (Slide4N), and RQ(d) to Chapter 6 (MACEDON). Below are the main research questions investigated:

RQ(a) How does interactive human–AI collaboration support programmers in understanding code by enabling them to inspect, refine, and validate AI-generated interpretations?

RQ(b) How does AI assistance support programmers in explaining code and results for different communicative purposes (e.g., technical reporting, stakeholder updates, and decision support)?

RQ(c) How do AI-generated presentation artifacts support programmers in sharing their work with stakeholders and colleagues?

RQ(d) How does AI-driven feedback support programmers in evaluating and optimizing code quality within existing development workflows?

1.2 Contributions

To answer these RQs, my dissertation makes contributions in three categories: *artifact contributions* (interactive systems and tools), *dataset contributions* (benchmarks and annotated corpora), and *empirical contributions* (evidence from user studies and experiments).

These three categories serve complementary roles in addressing the research questions. Artifact contributions provide interactive systems that let us test whether human–AI collaboration can support each workflow stage in practice. Dataset contributions supply the training data and evaluation benchmarks that ground the AI components of these systems. Empirical contributions provide evidence of how programmers use the systems and whether the collaboration improves their workflow. Together, the three categories form a complete research cycle: datasets enable models, models power interactive tools, and user studies evaluate whether the tools help programmers.

The contributions are organized by category below.

Artifact contributions. These are the interactive systems developed for each workflow stage. Each system implements the Propose–Inspect–Refine loop and addresses one or more research questions.

1. **HAConvGNN** (addresses RQ(a), understanding stage). A model and interactive interface that support human–AI collaboration in cross-cell code comprehension, enabling programmers to refine and validate AI-generated interpretations before they feed into documentation.

2. **Slide4N** (addresses RQ(c), presentation stage). A human–AI system that generates presentation artifacts directly from notebooks while keeping authors in control of scope, structure, and edits, reducing manual assembly work.

3. **MACEDON** (addresses RQ(d), evaluation and optimization stage). An IDE extension that supports human–AI collaboration in real-time, multi-dimensional code quality evaluation within existing development workflows, surfacing trade-offs and inviting iterative human decisions.

Dataset contributions. These provide the data foundations that enable and evaluate the AI components in the artifact contributions above.

4. **notebookCDG and NBDESCRIB** (supports RQ(a) and RQ(b), understanding and documentation stages). Two datasets addressing different workflow stages: the **notebookCDG** dataset for the *understanding* stage (cross-cell code-documentation pairs that train and evaluate HAConvGNN), and the **NBDESCRIB** benchmark with purpose-driven guidelines for the *documentation* stage (faithful descriptions grounded in both code and tabular outputs, addressing different communicative purposes). NBDESCRIB

also includes an interactive human–AI writing tool that supports collaborative documentation.

Empirical contributions. These provide evidence that the artifact and dataset contributions help programmers in practice.

5. **User studies and experiments** (spans all four RQs). Evidence from controlled studies, user studies, and case studies when humans collaborate with AI across the key workflow stages: understanding, documentation, presentation, and evaluation and optimization. Findings include: (i) formative studies and user evaluations on how programmers benefit from AI-assisted documentation during the understanding stage; (ii) benchmarks and error analyses revealing failure modes of current models for faithful description in the documentation stage; (iii) usability and outcome studies on human–AI slide generation in the presentation stage; (iv) evaluations of multi-dimensional code-quality feedback in the evaluation and optimization stage. Together these yield gains in efficiency, output quality, and user preference, plus design implications for future human–AI programming tools.

1.3 Dissertation Outline

The remainder of the thesis is organized as follows (see also Figure 1.2).

In Chapter 2, I review related work and background. I survey existing approaches organized by workflow stages: code understanding, documentation, presentation, and code evaluation and optimization. I also discuss the theoretical foundations of human–AI collaboration and interactive system design that inform the design decisions across all four projects.

In Chapter 3, I present HAConvGNN, which supports the *understanding* stage through context-aware documentation that spans multiple code cells in computational notebooks. I discuss how structured code analysis and hierarchical attention mechanisms improve programmers' ability to understand code, and how the resulting human-AI collaboration workflow enables refinement and validation of AI-generated interpretations.

In Chapter 4, I introduce NBDESCRIB, which supports the *documentation* stage through an interactive human–AI collaboration system for collaborative writing. To enable this system, the chapter first presents a task formulation and benchmark for purpose-driven documentation generation from code and its execution results, taking tabular output

as a common example; it then shows how explicit guidelines for different communicative purposes improve generation quality and representation.

In Chapter 5, I present Slide4N, which supports the *presentation* stage through a human–AI system for creating presentation slides from computational notebooks while maintaining author control. Building on understanding and documentation support, this chapter shows how to transform notebook content into presentation artifacts without requiring manual assembly work.

In Chapter 6, I present MACEDON, which supports the *evaluation and optimization* stage through an IDE extension offering real-time, multi-dimensional feedback on code quality during development via human-AI collaboration. Shifting from notebook-based authoring to IDE-based evaluation and optimization, this chapter demonstrates how continuous feedback on clarity, documentation, redundancy, and efficiency can guide code improvement and optimization.

In Chapter 7, I conclude with a summary of contributions, a discussion of limitations, and an outline of future research opportunities that emerge from this work.

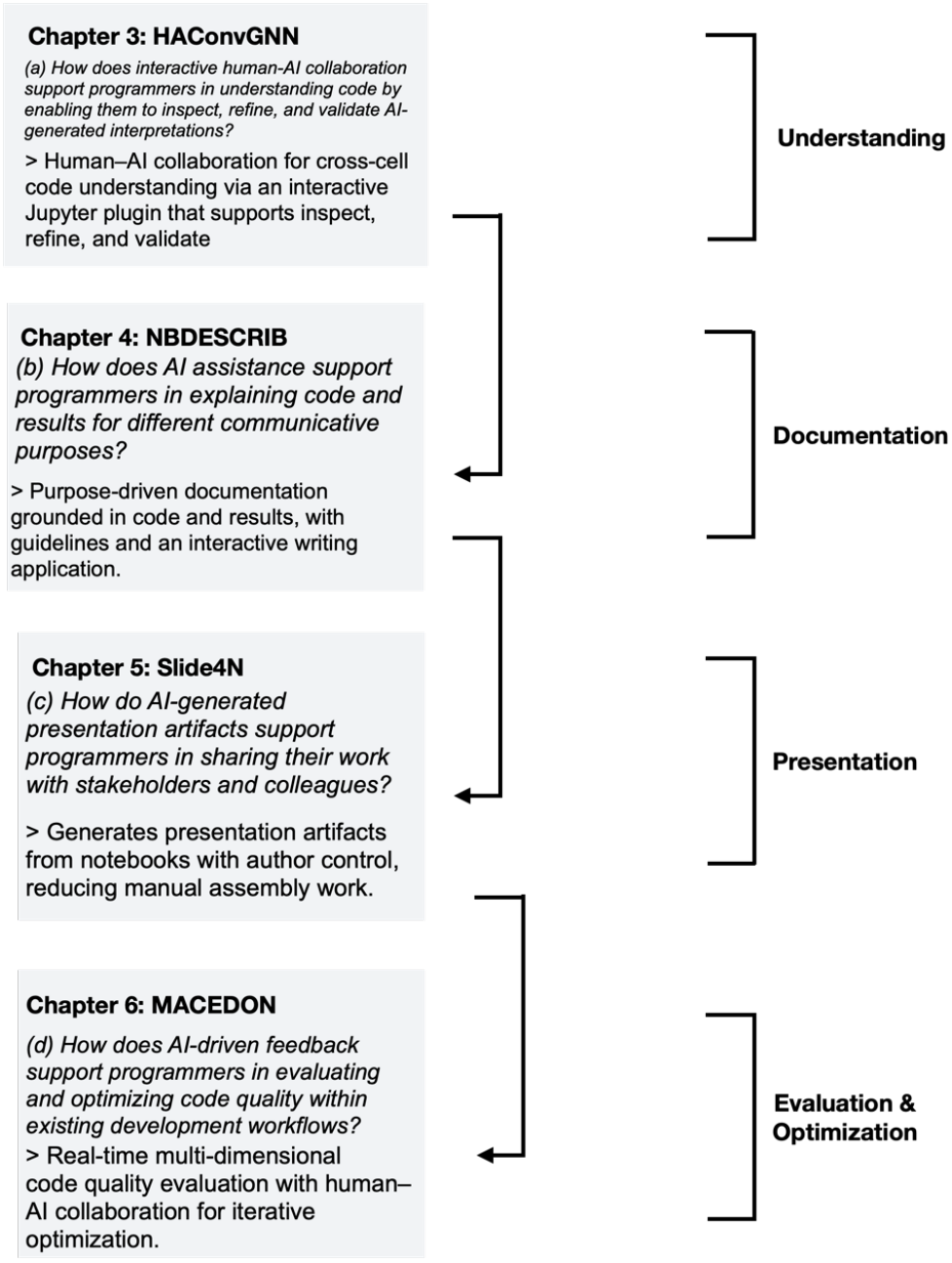


Figure 1.2: Chapters with main research questions (italic), main results, and research path connecting each chapter. Arrows show the dependencies and motivation that lead to the next research question.

Chapter 2

Related Work

This chapter reviews related work from two central perspectives: programming workflows and human-AI collaboration. It characterizes the multi-stage nature of how programmers work, establishes theoretical foundations for why human-AI collaboration can support these workflows, and surveys existing tools that address different stages. The review identifies gaps that motivate the four projects presented in this dissertation.

2.1 Programming Workflows

Programming is not a single task but a recurring, multi-stage process [96, 138]. Programmers work in diverse environments: computational notebooks for exploratory analysis, full-featured IDEs for software development, lightweight editors for scripting, and hybrid setups that combine multiple tools. Across these settings, they move among understanding code, documenting artifacts, evaluating and optimizing quality, and presenting outcomes for others. Each stage presents distinct challenges that shape how they interact with tools and artifacts.

Understanding code requires reconstructing mental models of system behavior and dependencies, especially when working with codebases they did not write [96]. In traditional source files, this involves navigating functions, classes, and modules; in notebook-style environments, understanding often spans multiple interleaved code cells, markdown, and execution outputs [90, 152, 167]. *Documenting* preserves the rationale behind implementation choices (sometimes called "design intent" [26]) and supports collaboration across codebases, APIs, and notebooks, but maintaining explanations alongside evolving code is

difficult [26, 167]. Empirical studies report that well-documented projects receive more external contributions and that documentation quality correlates with reduced onboarding time for new team members [3]. *Evaluation and optimization* involves correctness, debugging, efficiency, and readability [18], yet tools typically surface one dimension at a time or require separate analysis phases [155]. *Presenting* outcomes for stakeholders demands translating technical work into understandable forms [101, 221], whether through reports, slides, demos, or shared artifacts; creating such materials is laborious and often disconnects from the primary development environment [27].

These workflow challenges motivate tools that support programmers across stages and environments while fitting into existing practices. The next section establishes why human-AI collaboration is a suitable design paradigm for such support.

2.2 Benefits of Human-AI Collaboration in Programming

Human-AI collaboration positions the programmer as the decision-maker while AI proposes drafts, summaries, or feedback for human inspection and refinement. This paradigm aligns with how programmers already work: they iterate, verify, and steer outcomes rather than delegating entirely to automation [185]. Theoretical frameworks from cognitive science and learning research justify why such collaboration and multiple representations can improve programming workflows.

Cognitive load theory suggests that working memory is limited and that presentation affects learning and problem-solving [179]. Multiple representations can reduce extraneous load by matching information form to task; for example, combining code (verbal) with execution outputs (visual) reduces effort to understand behavior. The multimedia principle [128] and dual-coding theory [145] similarly support combining text, code, and visuals. Human-AI collaboration amplifies these benefits: AI can generate candidate explanations or descriptions, while humans verify, correct, and adapt them for context, keeping outputs grounded in intent.

Fully automated tools often fall short because they operate in isolation, require manual coordination, or produce outputs that lack context or introduce errors [26, 185]. Interactive collaboration addresses these limits by maintaining a feedback loop where humans steer and refine AI suggestions. The next section reviews existing tools through this workflow and human-AI collaboration lens, identifying gaps that motivate this dissertation's projects.

2.3 Existing Tools and Identified Gaps

This section reviews existing tools through the lens of programming workflows and human-AI collaboration. Tools are organized by four workflow stages: code understanding, documentation, presentation, and code evaluation and optimization. Each subsection identifies gaps where interactive human-AI collaboration could better support programmers.

2.3.1 Code Understanding

Programmers work in diverse environments, from computational notebooks for exploratory analysis to full-featured development environments for software engineering. Understanding code in either setting requires reconstructing mental models of behavior and dependencies.

Code summarization and comprehension tools provide foundations for understanding. Most existing datasets contain one summary per code snippet; for instance, CodeSearchNet [73] contains two million function-documentation pairs across six programming languages. Previous work focuses on generating summaries for single standalone snippets [7, 71, 75], exploiting the abstract syntactic tree (AST) information to capture relations between code elements [7, 71], and proposing graph-to-sequence models to learn node embeddings [33, 105, 214]. However, these approaches do not address the notebook context where a single markdown cell often explains multiple preceding code cells. In notebook-style environments, related code cells work together to accomplish a task, creating cross-cell dependencies. Existing methods cannot handle this multi-cell context. HAConvGNN addresses this gap through human-AI collaboration: it models hierarchical relationships between code cells and generates documentation that considers multiple cells, while a notebook plugin lets programmers inspect, refine, and validate the generated explanations.

2.3.2 Documentation

Whether code lives in notebooks or traditional source files, documentation bridges the gap between implementation and its consumers. It must often cover both the code and its outputs, especially when execution produces tables or visualizations.

Generating descriptive text from code and its outputs is essential for sharing or reuse [26, 221], yet many practitioners neglect to write appropriate descriptions, especially for code outputs like tables [160]. Existing literature explores code documentation [8, 125, 161] and table-to-text generation [111, 123, 146, 191] independently. Most datasets contain

one text per table or code on a specific topic [74, 174]. Table-to-text work focuses on standalone tables [28, 107, 116, 142, 147, 208]; code description models [4, 47, 59, 74, 80, 87, 200] focus on code alone, often without covering facts from execution outputs. In practice, documentation relates to both code and its table output simultaneously, and the same code-table pair can serve different purposes under different user requirements. Existing work does not consider joint context or varied purposes. NBDESCRIB addresses this gap with a dataset and task formulation for joint code-table description, along with guidelines for faithful text. An interactive writing tool supports human-AI collaboration, allowing practitioners to ground descriptions in evidence and revise for different audiences.

2.3.3 Communication and Presentation

Sharing results often requires moving beyond the primary development artifact. Data science teams present analyses to stakeholders; software teams prepare reports or demos. Creating presentation materials from code and its outputs is laborious and error-prone when done manually.

Computational notebooks weave code, documentation, and outputs together [100, 152], but have limitations for communication with non-technical stakeholders [27, 90]. Data science teams require frequent communication [41, 101, 221], and creating slides from notebooks remains laborious and repetitive. Some research improves notebook readability by simplifying them: Head *et al.* [67] proposed code gathering techniques to reorganize related cells; StickyLand [205] allows users to freely organize code. However, these tools still produce notebooks and do not generate presentable content for non-technical audiences. Other work supports slide generation directly from notebooks. RISE [10], an extension for computational notebooks, turns selected cells into slides exportable via nbconvert [84], but it transforms content as-is without analyzing or distilling key points. NB2Slides [230] applies a template-based approach, though it requires few excess cells, high-quality documentation, and a complete workflow (difficult in exploratory analysis) and lacks adequate user interaction support. Slide4N bridges this divide with human-AI collaboration: authors select cells of interest, the system proposes slide content and layout, and authors refine incrementally while maintaining control.

2.3.4 Code Evaluation and Optimization

Beyond authoring and documenting, programmers must evaluate code quality and optimize performance. This need arises across environments, whether in lightweight editors or full-

featured development tools, as programmers balance correctness, runtime, memory, and readability.

Traditional approaches include performance analysis tools (e.g., HPCToolkit [2], Speedoo [34]) that collect runtime data to identify bottlenecks, and automatic performance regression detection [92]. These operate as separate analysis phases, requiring programmers to compile, execute, and analyze outside the development workflow. Static analysis and refactoring tools [183, 219] and capabilities integrated into development environments [76] offer automated transformations and style enforcement based on predefined rules [136, 144], but lack contextual understanding for complex optimization. Neural program repair [186] and learning-based optimization [169] demonstrate AI-driven enhancement. Mixed-initiative research [14, 213] explores AI assistance, but focuses on code generation rather than evaluation. Recent workflows leverage LLMs for generating and optimizing code [1, 30, 51, 141], and tools like GitHub Copilot assist with completion and optimization [14, 140]. Programmers still need real-time insights into their code's state across multiple dimensions before crafting optimization strategies [155]. MACEDON addresses this gap through human-AI collaboration: an extension for development environments provides real-time, multi-dimensional evaluation (clarity, documentation, redundancy, runtime, memory), making trade-offs visible and inviting programmers to steer optimization decisions as they code.

Chapter 3

Context-Aware Code Understanding across Multiple Code Cells

This chapter addresses **RQ(a)** from the introduction, focusing on the *understanding* stage of programming workflows. Our main contribution is an interactive human–AI collaboration system, a Jupyter plugin, that drafts context-aware documentation and lets programmers inspect, refine, and adopt the text to support code understanding. To enable this system, we first construct the notebookCDG dataset and develop the HAConvGNN model; these preparatory components provide the generation capability that the collaboration interface builds upon. We then present the plugin and a user study validating that programmers benefit from the generated content when understanding multi-cell code.

3.1 Motivation

This chapter addresses the *understanding* stage of the programming workflow introduced earlier (Figure 1.1). Programming is a multi-stage process: before documentation, communication, or optimization, programmers first need to understand what the code is doing, why it was written, and how different parts connect as the work evolves. When code is spread across many small pieces and revised over time, this understanding work becomes a real barrier to iteration, review, and collaboration.

We frame code understanding as a human–AI collaboration problem. Our goal is an interactive system that drafts context-aware explanations and lets programmers inspect, edit, and refine the text so that it matches their intent and the current task. To build this

system, we first lay the groundwork: the dataset we construct and the model we develop serve as preparatory components that enable the downstream human–AI collaboration interface (a Jupyter plugin). We study the setting where code is spread across multiple cells, which makes context-awareness essential and provides a practical environment for both data collection and evaluation.

Many existing automated approaches treat summarization as a single output for a single snippet. In multi-cell settings, however, useful explanations often depend on cross-cell dependencies and on what the reader is trying to do at that moment. A fixed summary can miss key context, emphasize the wrong details, or read as reasonable while still failing to support the programmer's next step. Fully manual writing is also not a complete answer, because documentation is often skipped during exploration and is hard to keep up to date [167].

With this goal, we address research question (a): how to generate context-aware documentation that goes beyond basic description for multi-cell projects, and whether it helps programmers during programming. The dataset and model we present in the following sections pave the way for the interactive human–AI collaboration tool described later.

In this chapter, we use the term *multi-cell code* to refer to a sequence of two or more adjacent code cells in a computational notebook that collectively accomplish a task and are described by a single documentation (markdown) cell. This structure is common in Jupyter-style notebooks, where a markdown cell often precedes or follows a group of related code cells.

Multi-cell awareness is important for documentation generation because the meaning of an individual cell often depends on variables, data structures, or operations defined in neighboring cells. A model that considers only one cell at a time may miss these dependencies and produce incomplete or misleading documentation. Our ablation study (Section 3.5) confirms this: removing the hierarchical attention that connects cells leads to a substantial drop in ROUGE scores.

In recent years, computational notebooks such as Jupyter have become popular platforms for data scientists to document ideas, write code, and visualize results in a single document [187]. Documentation in a notebook often covers multiple code cells and explains both logic and outcomes.

As preparatory steps toward the human–AI collaboration interface, we first address the code documentation generation (CDG) task. Since there is no publicly available CDG dataset for multi-cell settings, we construct a new dataset (**notebookCDG**) which contains around 28k processed code-documentation pairs extracted from 2,476 highly-ranked notebooks from Kaggle competitions (details in Section 3.3)

Documentation	
ground truth	Implementing Neural Network
our Model	Implementing Neural Network
code2seq	The following function of the model
graph2seq	After perturbations
T5-small	Model

Code Cells

```
import keras
from keras.utils import plot_model
from keras.models import Model,Sequential,load_model
...
```

```
def nn_model(X,y,optimizer,kernels):
    input_shape = X.shape[1]
    if(len(np.unique(y)) == 2):
        op_neurons = 1
        op_activation = 'sigmoid'
        loss = 'binary_crossentropy'
    else:
        op_neurons = len(np.unique(y))
        op_activation = 'softmax'
        loss = 'categorical_crossentropy'
    classifier = Sequential()
    ...
    classifier.summary()
    return classifier
```

```
model = nn_model(X_train,y_train,'adam','he_uniform')
history = model.fit(X_train, y_train, batch_size = 64, epochs = 1000,
    validation_data=(X_test, y_test))
```

```
pd.DataFrame(abs(train.corr()['Survived'])
.sort_values(ascending = False))
```

Table 3.1: An example of multiple code cells after one documentation block

Previous literature has explored techniques to generate documentation for software

code snippets one at a time [65, 66, 105, 214]. However, in computational notebooks, one documentation (in a *markdown cell*) can cover more than one code cells after it. For instance, the ground truth text in Table 3.1 is a single documentation covering four code cells. Existing work on CDG [7, 71, 75, 89, 105] does not consider such structure information since they only focus on documentation generation for single code snippet (i.e., one function, or one expression).

To account for the above mentioned properties of documentation in computational notebooks, in this chapter, we propose a graph-augmented encoder-decoder model to generate documentation for notebooks (Section 3.4). In particular, our model consists of three parts: a code sequence encoder, an auxiliary documentation text encoder based on the already predicted documentation tokens, and a Hierarchical Attention-based Convolutional Graph Neural Network (HAConvGNN) component.

The first two sequence encoders encode the semantic information in code and documentation text, respectively. The graph encoder encodes the contextual abstract syntactic trees (i.e., AST extracted from the code sequence). In order to capture the relations between code sequences and the corresponding text documentations, we further employ a hierarchical attention mechanism consisting of a low-level attention module and a high-level attention module. The former attends to the token in a code sequence and the latter attends to the corresponding code cells in the AST tree.

Experiments show that our model achieves better performance on the *notebookCDG* dataset compared to baseline models on ROUGE scores, and in a multi-dimensional human evaluation study.

Based on these results, we built a downstream Jupyter plugin [192] to further validate research question (a). In the follow-up user study [188], participants reported that the automatically generated documentation reminded them to document code they would have skipped, and improved their satisfaction with their notebooks.

In summary, this chapter advances **RQ(a)** by supporting the *understanding* stage through interactive human–AI collaboration. The main contribution is the Jupyter plugin as the interactive human–AI collaboration system; a downstream user study validates that programmers benefit from the generated content for understanding their code. The notebookCDG dataset and HAConvGNN model serve as preparatory work that enables context-aware documentation generation, paving the way for this collaboration interface. The experiment code and data are shared[1].

[1] https://github.com/xuyeliu/HAConvGNN

3.2 Related Work

	Overall	Train	Dev	Test
Notebooks number	2,476	2,426	1,390	1,394
Code-documentation pairs	28,625	22,851	2,856	2,856
Code vocabulary size	20,522			
Code AST vocabulary size	67,211			
Documentation vocabulary size	13,053			
Avg. # token in documentation	9.15	9.13	9.37	9.18
Max. # token in documentation	202	202	130	104
Std. # token in documentation	8.40	8.44	8.27	8.25
Avg. # token in code cell(s)	65.38	65.50	65.41	64.39
Max. # token in code cell(s)	400	400	400	395
Std. # token in code cell(s)	68.93	69.16	68.23	67.71
Avg. # token in code AST	181.08	181.47	180.77	178.24
Max. # token in code AST	1732	1548	1732	1167
Std. # token in code AST	192.19	193.00	190.43	187.40

Table 3.2: ***notebookCDG*** dataset statistics. The overall code-to-markdown ratio is 2.2195, which suggests one markdown corresponds to more than one code cells.

As Section 2.3.1 establishes, existing code summarization focuses on single snippets and does not address the notebook context where one markdown cell often explains multiple subsequent code cells. We extend this line of work to notebook multi-cell documentation by introducing a new dataset and model.

Prior work on code summarization provides technical foundations we build upon. [75] used attention-based seq2seq models for single-snippet summarization. Hu et al. [71] and Alon et al. [7] exploited abstract syntactic tree (AST) information to better capture relations between code elements. [214] and [33] proposed graph-to-sequence models that learn node embeddings and reassemble them into graph embeddings. However, these approaches cannot represent the hierarchical structure of multiple adjacent code cells that share one documentation cell in notebooks.

In our notebook CDG task, multiple adjacent code cells may correspond to one documentation, and these cells form a hierarchical structure. We represent this structure as a hierarchical AST graph [94] and propose Hierarchical Attention-based Convolutional Graph Neural Network (HAConvGNN) to handle it. Our new dataset (*notebookCDG*) is designed for this setting, where a documentation text can correspond to several code snippets, as reflected in the statistics in Table 3.2.

3.3 *notebookCDG* Dataset

Code documentation generation (CDG) for notebooks is a relatively new task. To our best knowledge, we could not find an appropriate dataset for this task. Thus, we decided to construct a new dataset and share it with the community. Publicly shared notebooks on Github are often ill-documented [167], thus are not suitable for constructing the training dataset for the CDG task. A recent work [187] manually analyzed 80 publicly available notebooks on two Kaggle challenges (i.e. out of 12,000 notebooks submitted to Titanic and HousePrice). Kaggle allows community members to vote up and down on those notebooks, and [187]'s findings show that the highly-voted notebooks are of good quality and quantity in code documentation. Inspired by their work, we decided to utilize the top-voted Kaggle notebooks to construct the **notebookCDG** dataset[2].

We collected the top 10% highly-voted notebooks from the top 20 popular competitions on Kaggle (e.g. Titanic). We checked the data policy of each of the 20 competitions, none of them has copyright issues. We also contacted the Kaggle administrators to make sure our data collection complies with the platform's policy. In total, we collected 3,944 notebooks as raw data.

3.3.1 Data Preprocessing

We performed various preprocessing steps to prepare the dataset, following [104]. For example, we removed notebooks in non-English language. One major difference between our dataset and previous datasets is that in previous datasets, each documentation unit corresponds to one code snippet, whereas in our dataset, one documentation unit may correspond to up to four code snippets (code cells). We first located the markdown cells that

[2]We share the notebookCDG dataset with processed 28k code-document pairs at https://ibm.biz/Bdfpk6

have code cells beneath them. According to [187], there are nine categories of documentations in a notebook, some are related to code, some are not related to code. For those types closely related to code (`Process` and `Headline`), which take up 80% of the cases, we can directly use the markdown cell as documentation. For some other types, such as the `Result` type, which interprets the rendered result table or plot thus are often long and irrelevant to the code, we used a list of result-indicating keywords (e.g., "shows," "indicates," "demonstrates") to filter out the key sentences from the markdown cell as the documentation. Other special types of documentation include `Reason` and `Education`, which also use long word sequences to explain why the author did something. In these cases, based on our observation, we used the first sentence as the documentation, as the first sentence is often related to the code cells.

Our analysis shows that for one markdown cell, there are at most four code cells following a given markdown cell. We construct our dataset to have a structure with one documentation unit and four code sequence units, and fill with empty sequences if there are fewer than four code sequences. As part of the data preparation, we also parse each of code sequence to an AST graph structure through a Python AST library[3]. While doing so, we removed all the non-Python notebook magic (e.g. %matplotlib).

3.3.2 Dataset Core Statistics

After data preprocessing, the final dataset contains 2,476 notebooks out of the 3,944 notebooks from the raw data. It has 28,625 code–documentation pairs. The overall code-to-markdown ratio is 2.2195, which suggests one markdown corresponds to more than one code cells. Then, the code-documentation pairs are randomly split into train, dev, and test subsets, following a 8:1:1 ratio (Table 3.2).

Our *notebookCDG* dataset has a vocabulary size of 13,053 for the documentation sequence, a vocabulary size of 20,522 for the code sequence, and 67,211 for the parsed code AST node. On average, each pair of code-documentation has 65.38 code tokens, and 9.15 documentation tokens. When code is translated to AST structure, on average it has 181.08 tokens.

[3] https://docs.python.org/library/ast.html

3.4 Approach

Our model is built upon the standard encoder-decoder structure. To handle multiple code cells in computational notebooks, we propose a hierarchical attention mechanism based on convolutional graph neural network (HAConvGNN) for capturing the relevant code cells during the decoding stage.

The system architecture is illustrated in Figure 3.1. Below, we describe each module in detail.

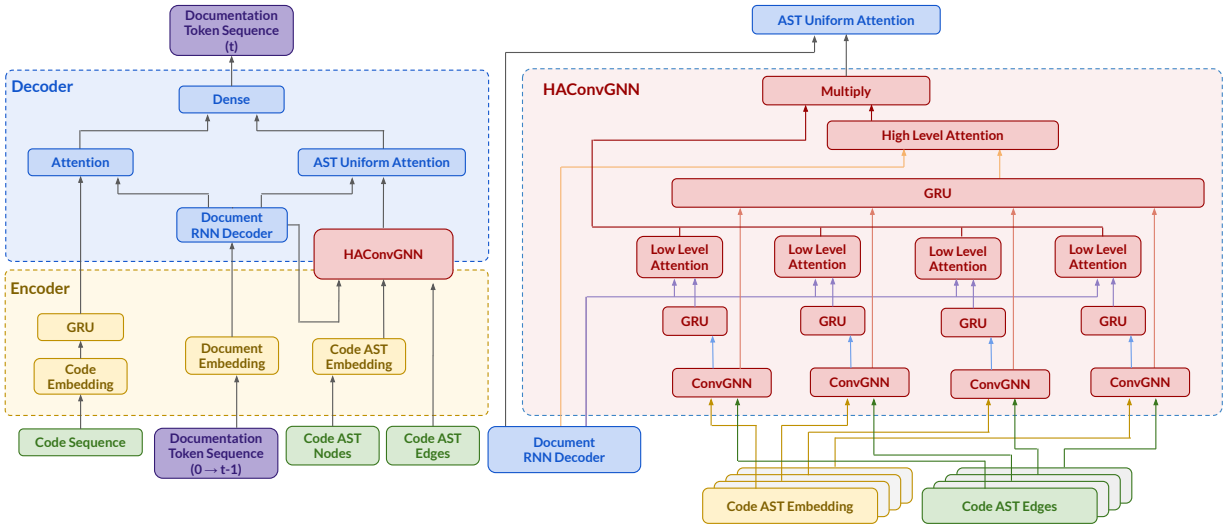


Figure 3.1: HAConvGNN model architecture

3.4.1 Model Input

As mentioned in Section 3.3, we found that there are up to four adjacent code cells under a markdown cell, thus we constructed the *notebookCDG* dataset to have one documentation mapping to four code cells, and used empty code cell as padding. Therefore, when generating the abstract syntactic tree (AST) for a code cell, we can assemble up to four AST trees into a higher level graph structure.

In summary, each training data point has four parts: the tokenized code sequence, the tokenized documentation sequence, the nodes of the AST graph generated from the code sequence, and the edges (topology) of the AST graph generated from the code sequence.

We denoted code sequence input as $S = \{s_1, s_2, ..., s_n\} \in \boldsymbol{S}$ where s_i is sequence consisting of a sequence of code token embeddings $s_i = \{w_1, w_2, ..., w_k\} \in \boldsymbol{W}$ in which $\boldsymbol{W}$ is the token embedding space and k is the length of s_i. Next we construct the AST graph input $A = (V, E)$ where V are the nodes containing the original code, E are the edges which denote whether two nodes are connected or not in the AST graph.

3.4.2 Embeddings

We use three embedding layers to generate embeddings for the tokenized code sequence, the nodes in an AST graph, and the documentation decoder, respectively.

3.4.3 Encoder

We use one encoder to encode the source code sequence, and additional four encoders to encode up to four code cells' AST graphs. In addition, we have a high-level GRU encoder layer for all the four AST graphs to generate one high-level output. More specifically, the encoder for the tokenized code sequence is a GRU with an output length of 256. An AST graph encoder is a collection of Convolutional Graph Neural Networks layers followed by a GRU layer of output length 256. We use four AST graph encoders for up to four code cells. Following [105], the number of hops in our GNN layers is set to 2.

3.4.4 HAConvGNN

The key design of our HAConvGNN model is the hierarchical attention. When handling AST graphs input, instead of blending these 4 code cells as a whole sequence, we propose to use a hierarchical attention mechanism (low-level attention and high-level attention in HAConvGNN in Figure 3.1) on these AST graphs to better preserve the graph structure.

Firstly, the four code cells' AST graph can be represented as $G = \{G_1, G_2, G_3, G_4\}$. We denote the decoder output (i.e., the predicted documentation tokens up till $t-1$) as $D \in \mathbb{R}^{n \times d}$ where d is the dimension. We further denote each code cell's AST graph as $G_i \in \mathbb{R}^{m \times d}$ where m is the number of nodes. After using a high-level encoder to encode the AST graph input, we execute a graph-level attention to get high-level attention score:

$$\alpha(G_i, D) = DG_i^\top / \sqrt{d} \tag{3.1}$$

Then we apply softmax on α, given by:

$$b^i = \frac{exp(\alpha(G_i, D))}{\sum_j exp(\alpha(G_j, D)))} \tag{3.2}$$

In this way, we get the results denoted as $\alpha = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$. This is our high-level attention weights indicating the relations between each code cell and the already predicted documentation sequence D.

Secondly, we apply an attention mechanism on each code cell to find the relations between nodes in a code cell's AST and the predicted documentation sequence D. For each code cell's AST tree $G = \{G_1, G_2, G_3, G_4\}$, we apply the same operation as in EQ.1 and EQ.2. As a result, for each code cell G_i, we are able to get a new low-level attention weight vector β_i over its m nodes. For all four code cells, we denote these attention scores as $\beta = \{\beta_1, \beta_2, \beta_3, \beta_4\}$, where each β_i corresponds to the weights for the nodes in cell i.

Eventually, we fuse these attention weights (α and β) with code cells:

$$O = \sum_{i=1}^{4} \alpha_i \sum_{j=1}^{m} \beta_{i,j} G_{i,j} \tag{3.3}$$

Now we get the AST matrices from HAConvGNN. It is then concatenated with code matrices into a single context matrix. Note that code matrices are based on the code sequence input with a separate uniform attention (see the left "Code Sequence" in Figure 3.1). Next, we apply a linear projection to project the merged context matrix into a 256 dimension space. This is an effective way to avoid overfitting during the training process. Finally, we flatten the new context matrix and apply another linear layer to project it into an output. The output layer size is the vocabulary size. By applying the Argmax function to the output layer, we can obtain the predicted next token (i.e., documentation token at time step t) in the output sequence.

3.5 Experimental Setup

3.5.1 Implementation Details

We split our dataset into training, development, and test datasets at a 8:1:1 ratio. We use the Adam optimizer [93] with a batch size of 20. The learning rate is 0.001 and the code sequence embedding size is 100. In the encoder, we use GRU [36] with the hidden size of 256. The hop size of our GNN is 2. The dropout rate of our attention layer is 0.5.

3.5.2 Baselines

We compare our model against two baseline models for single code snippets.

Models	ROUGE-1			ROUGE-2			ROUGE-L		
	P	R	F1	P	R	F1	P	R	F1
Baselines									
code2seq	11.45	8.46	8.23	1.67	1.11	1.11	13.13	10.28	10.24
graph2seq	13.21	9.87	9.51	2.86	1.99	2.03	14.46	11.40	11.18
Our Model & Ablation Study									
HAConvGNN (Our Model)	**22.87**	**16.92**	**16.58**	**6.72**	**4.86**	**4.97**	**24.03**	**18.60**	**18.54**
HAConvGNN with low-level attention without high-level attention with uniform attention	20.66	15.65	14.91	4.74	3.92	3.80	21.84	17.27	16.81
HAConvGNN with low-level attention without high-level attention without uniform attention	19.57	14.59	14.23	4.87	3.56	3.63	20.83	16.24	16.12
HAConvGNN without low-level attention without high-level attention with uniform attention	11.39	7.73	7.82	1.58	1.06	1.08	13.13	9.47	9.82

Table 3.3: ROUGE scores for the baselines, our model, and the ablation models. Results show that our model has higher scores for all three metrics, demonstrating a robust advantage over the code2seq and graph2seq models.

code2seq. [7] proposed a *code2seq* model to generate a summary for a C# function. The model creates a vector representation for each AST path separately through an encoder. During decoding, the model uses attention to select the relevant paths. We re-implement this model and apply it on our dataset.

graph2seq. [214] proposed a graph-to-sequence learning framework that maps an input graph to a sequence of vectors and uses an attention-based LSTM method to decode the target sequence from these vectors. The authors tested the model on natural language question generation from the SQL query task. We re-implement this model using all rec-

ommended parameters from the original paper.

3.5.3 Experimental Details

The training time of code2seq model is around 2.5 hours per epoch; the training time of graph2seq is around 2.75 hours per epoch; the training time of T5-small is around 3.25 hours per epoch; the training time of our HAConvGNN model is around 2.65 hours per epoch.

The training environment of code2seq, graph2seq, and HAConvGNN is three GPUs using Parallelism. The training environment of T5-small is two GPUs.

code2seq and graph2seq are implemented in Keras framework[4]. T5-small model is implemented based on Huggingface repo [5].

3.6 Automated Evaluation

We use ROUGE scores [117] to evaluate our model's performance with regard to the ground-truth documentation content. We report ROUGE-1, ROUGE-2, and ROUGE-LCS (longest common sub-sequence). As shown in Table 3.3, our HAConvGNN model outperforms the other two baselines in all ROUGE metrics.

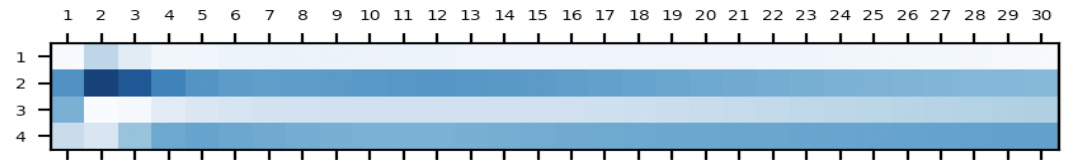


Figure 3.2: Attention visualization for the data point illustrated in Table 3.1. Each row represents a code cell, and each column is a code token. In this example, it shows the second and third token in the second code cell (“ nn_model”, “X”) contribute the most to the predicted documentation in Table 3.1.

[4] https://github.com/Attn-to-FC/Attn-to-FC
[5] https://github.com/huggingface/transformers

Ablation study. In order to better understand the impact of the attention components in our model, we also perform an ablation study (Table 3.3). Our ablation study evaluates how low-level attention, high-level attention, and AST uniform attention contribute to the model. More concretely, we generate ablation models as the following:

(1) without high-level attention in the hierarchical attention: we remove high level attention component in Figure 3.1 in our HAConvGNN structure. That means we do not compute attention weights for separated code cells.

(2) without AST uniform attention: we do not apply uniform attention mechanism (i.e., the attention component above *HAConvGNN* in Figure 3.1) for our HAConvGNN output with the decoder.

(3) without low-level or high-level attentions: we remove separated low-level attention components in Figure 3.1) in our HAConvGNN structure. Note that when we remove these separated attentions, we also remove the high-level attention (thus the entire hierarchical attention structure). We treat multiple code cells as a standalone code snippet in this situation and process graph data with the original GNN layer (see the last row in Table 3.3).



Figure 3.3: Average rated scores given by human evaluators to each method across three dimensions.

In general, we found that the hierarchical structure in our HAConvGNN is proven to enhance our final performance. It is worth noting that the separated attention mechanism is essential in our model. Remember that we use the attention mechanism for our four code cells separately. Treating them as a single big code snippet leads to a considerable performance drop (see the last row in Table 3.3). This demonstrates that the hierarchical structure in our model can better handle the code documentation generation task for multiple code cells.

Model	Correctness	Informativeness	Readability
Groundtruth	$\overline{x}$ = 1.09, σ=0.95	$\overline{x}$ = 0.85, σ=0.97	$\overline{x}$ = 1.03, σ=1.01
Our model	$\overline{x}$ = 0.21, σ=1.33	$\overline{x}$ = 0.17, σ=1.18	$\overline{x}$ = 0.67, σ=1.20
Code2seq	$\overline{x}$ = -0.59, σ=1.29	$\overline{x}$ = -0.72, σ=1.17	$\overline{x}$ = 0.03, σ=1.35
Graph2seq	$\overline{x}$ = -0.30, σ=1.40	$\overline{x}$ = -0.21, σ=1.25	$\overline{x}$ = 0.32, σ=1.35

Table 3.4: Human Evaluation Result

Attention Visualization. Our high-level attention mechanism can indicate the most relevant code cell when generating the documentation for several code cells. Figure 3.2 illustrates the attention heatmap for the code example in Table 3.1. Each row represents a code cell, and each column corresponds to a code token. It seems that the model pays more attention to the second code cell (especially the first few tokens) when generating the documentation "*Implementing Neural Network*".

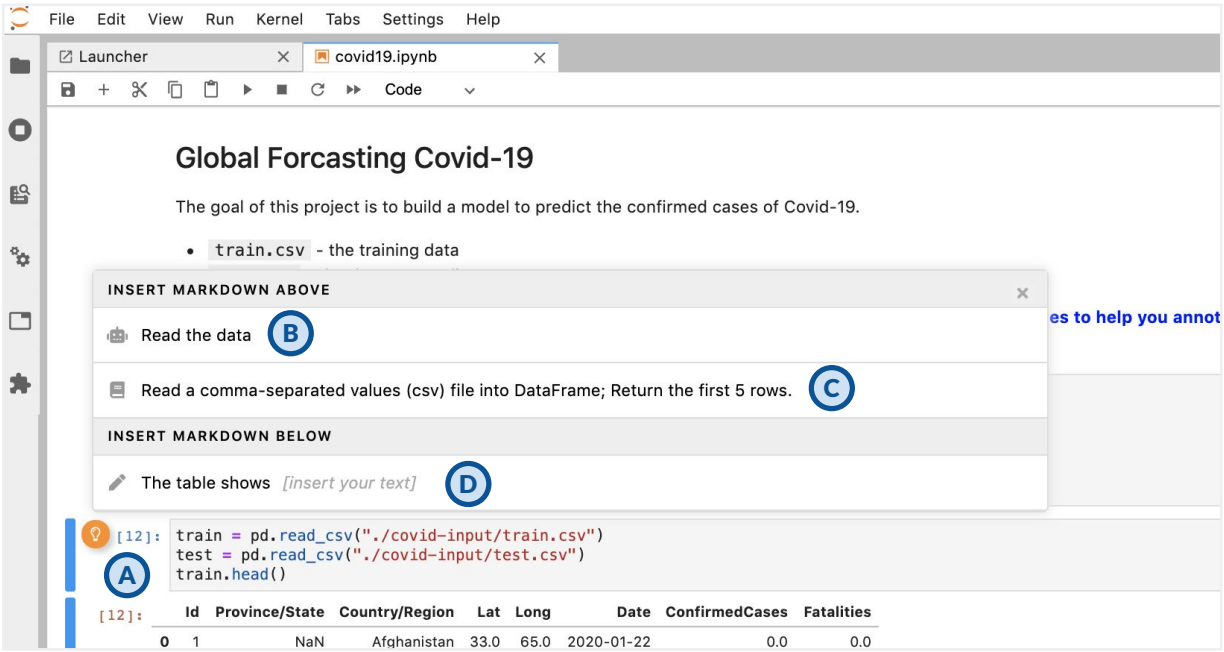


Figure 3.4: We implement a downstream application as a Jupyter Notebook plugin (A) to assist users documentation writing, incorporating the HAConvGNN-predicted results (B) next to an IR-based approach (C), and a user-prompt approach (D).

3.7 Human Evaluation

We also conduct a human evaluation to further evaluate our model against the two baselines and the ground truth.

Participants. Our human evaluation task involves reading code snippets and rating the generated documentation of the codes. We recruited participants with data science and machine learning backgrounds ($N = 15$).

Task. We randomly selected 30 pairs of documentation and code(s) from our dataset. Note that each pair has only one summary, but may have multiple code snippets. Each participant is randomly assigned 10 trials, and the order of these 10 trials is also randomized. Each pair is evaluated by 5 individuals. In each trial, a participant reads 4 candidate documentation for the same code snippet(s): three generated by the three models, and the other one is the groundtruth. Participants do not know which documentation text is from which model. The participant is asked to rate the 4 documentation texts along three dimensions using a five-point Likert-scale from -2 to 2.

- *Correctness*: The generated documentation matches with the code content.
- *Informativeness*: The generated documentation covers a greater number of relevant information units compared with other candidates.
- *Readability*: The generated documentation is in readable English grammar and words.

Evaluation Results. We conducted *pairwise t-tests* to compare each model's performance. Each pairwise comparison uses $df = 150$ (30 items $\times$ 5 raters). We report means, standard deviations, and significance levels below; full test statistics are available in the supplementary materials. The result (Table 3.4) shows that for the *Correctness* dimension, our model (avg=0.21) is significantly better than the other two baselines (avg=-0.59 for code2seq, avg=-0.30 for graph2seq, both $p<.01$). Our model is also the only model that has a positive rating. For the *Informativeness* dimension, groundtruth also has the best rating. Our model (avg=0.17) comes in second and outperforms code2seq (avg=-0.72, $p<.01$) and graph2seq (avg=-0.21, $p<.01$).

For the *Readability* dimension, in which we consider whether generated documentation is a valid English sentence or not, groundtruth outperforms all ML models again, but our model (avg=0.67) also significantly outperforms baseline models code2seq (avg=0.03

p<.01) and graph2seq (avg=0.32 p<.01). Our model can generate more readable documentation than baselines.

All the results suggest that our model has above-zero ratings, which suggests it reaches an acceptable user satisfaction along all three dimensions.

3.8 Comparison With Transformers

We also carried out an additional experiment to compare our model with T5 [158], which is a state-of-the-art transformer encoder-decoder model. In order to fairly compare our model against T5, we do not use any pre-trained embeddings for the T5 model. Also, T5 input has limitation for the input token length thus we did not feed AST hierarchy into it. More specifically, we initialize a T5-small model[6] with random weights and train this model using our training data. Our code adapts the transformer models from HuggingFace [110]. We use the dev dataset to choose the hyperparameters and evaluate the trained model on our test dataset. The ROUGE F1 scores for the trained T5-small model are as follows: ROUGE-1 = 17.55, ROUGE-2 = 4.57, ROUGE-L = 19.53.

We found that the trained T5-small model achieves slightly better results than our model in ROUGE-1 and ROUGE-L. In practice, we found that the T5-small model relies on a much more hyperparameters and tends to generate less informative content compared to other models (see the documentation generated from different models in Table 3.1 for an example).

But in our dataset, as reported in Table 3.2, the max AST token sequence is 1,732, which is too long as T5 input (512) or BART input (1,024). That is why T5 in Sec 8 can only take the raw code sequence as input, instead of the AST hierarchy. It is known that programming code has a tree-based hierarchy and leveraging such AST hierarchy can enhance the baseline model (e.g., [7]). Our contribution is that we provide a hierarchical attention architecture that is well suited for the programming code nature and can generalize to a much longer length of code inputs. Imagine in a scenario where we can feed a whole code repo as training input by treating each code file as a lower layer, and connecting them through function/variable referencing – our architecture can also handle that. In general, we think our model is orthogonal to the standard transformer models. One interesting future work is to integrate our hierarchical attention mechanism into the transformer-based structure instead of a GRU-based structure.

[6]In a pilot study, training a T5-base model (with random initialization) on our dataset leads to worse results.

3.9 Downstream User Application

Through the downstream Jupyter plugin, we further validate research question (a): whether programmers benefit from documentation during programming. We designed the plugin to assist document writing in data science programming (Figure 3.4).

The plugin is triggered when detecting users focusing on a code cell (Figure 3.4.A). The plugin then reads the contents from the focused cell and its adjacent cells, and sends the content to the backend. The backend server first generates a code summarization using the HAConvGNN model (Figure 3.4.B). In addition, we implemented two other approaches to generate documentation that was intended for explaining a design decision or explaining a technical concept for educational purposes. We retrieved the relevant documentation from the API webpage for educational purposes (Figure 3.4.C) and we used prompts to nudge users to explain an output (Figure 3.4.D). If the user likes one of these three candidates, they can simply click on one of them, and the selected documentation candidate will be inserted above the code cell (if it describes what and why for the code), or below it (if it interprets the result of the code).

Our plugin went through several rounds of pilot testing and iterative design. During pilot sessions, participants reported that the plugin reminded them to document code they would have ignored, and that it reduced the time they spent on documentation while exploring data science tasks. These observations are based on informal feedback collected during pilot testing; a formal evaluation is reported separately in [188].

3.10 Conclusion and Future Work

This chapter addresses **RQ(a)**. We establish that documentation helps programmers through a literature review and a downstream user study, and we address documentation generation with the notebookCDG dataset and HAConvGNN. Both automated and human evaluations show that HAConvGNN outperforms baseline models for the multi-cell documentation task. The downstream plugin study [188] confirms that programmers benefit from the generated documentation.

Through the downstream application, we observed that HAConvGNN and existing code summarization models still have limited ability in two areas. First, they focus on code structure and logic; they do not handle code that produces *output*—such as tables, charts, or data summaries—well. Second, programmers often need different kinds of descriptions in the computational notebook settings. Some focus on the output, which describe the

distribution of variables in a table, report max or min values, or analyze the results. Some describe future work or research directions. Some summarize what the code does when run, or explain why a particular algorithm was chosen. These different description categories call for different text-generation strategies, but current models are trained mostly on one style and struggle to cover all of them accurately.

To move forward, we built a new dataset and task (presented in Chapter 4) that explicitly includes code with output and supports multiple description purposes. We analyzed where state-of-the-art models fail and identified common error patterns, laying a reasonable foundation and a potential direction for the future research on this area.

3.11 Ethical Concern

Our task is an instance of natural language generation task, thus it may have potential risk and ethical issues similar to any other NLG tasks, such as the generated content may have offensive language. However, we believe our task and our approach has minimum risk of such ethical issues, due to two reasons: firstly, the language used in the context of machine learning code documentation is more strict to technical terms, offensive language is less likely to appear in the dictionary thus in our model; secondly, the dataset construction method is to use highly-voted notebooks from a publicly available Kaggle community, there is unlikely to have offensive languages in these highly-voted notebooks.

Chapter 4

Generating Documentation from Code and Results for Different Purposes

This chapter addresses **RQ(b)** from the introduction, focusing on the *documentation* stage of programming workflows. Once programmers have a sound understanding of their code, they often shift focus to how to describe execution results for different communicative purposes [167]. Code execution yields various types of results; tabular output is one common form that we adopt as the focus of this chapter. Our main contribution is an interactive human–AI collaboration system for collaborative writing that drafts purpose-driven descriptions from code and its execution results and lets authors guide, review, and refine the text. To enable this system, we first construct the NBDESCRIB benchmark and conduct model analysis; these preparatory components ground the generation capability and task formulation that the collaboration interface builds upon. While Chapter 3 supported code understanding through context-aware documentation across cells, this chapter addresses a complementary challenge: describing results and insights from code execution, with explicit guidelines for different communicative purposes.

4.1 Motivation

This chapter addresses the *documentation* stage of the programming workflow introduced earlier (Figure 1.1). Documentation often needs to describe both the code and the results it produces; execution results take many forms, and we focus on tabular output as a common and representative example. Our goal is an interactive human–AI collaboration system for

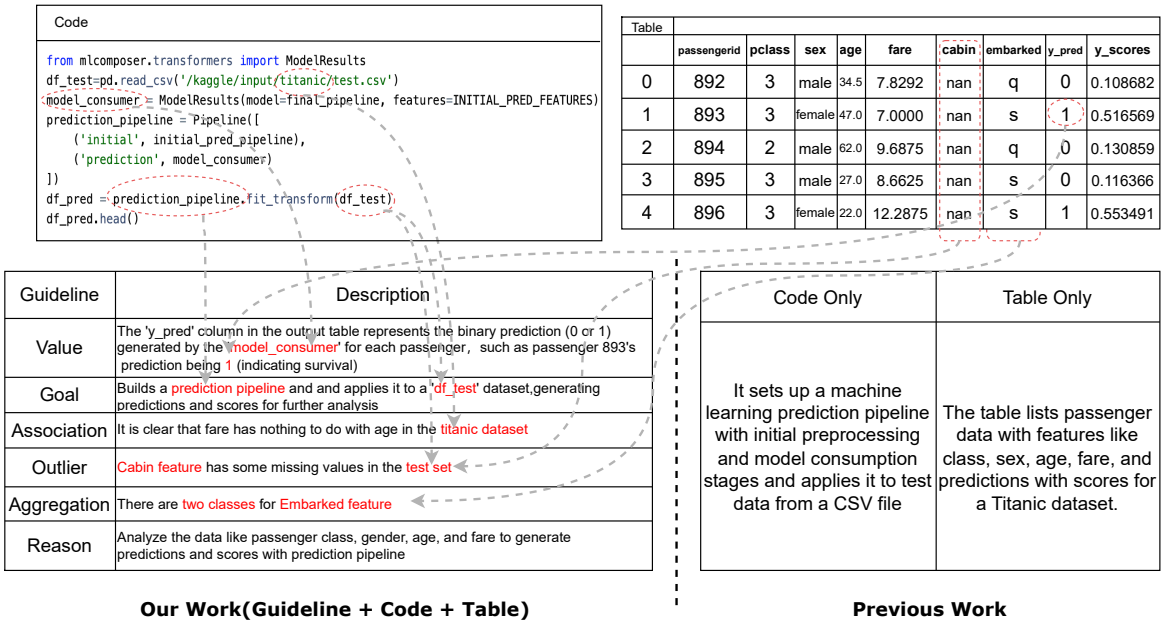


Figure 4.1: An example in our proposed NBDESCRIB dataset, which targets generating purpose-driven descriptions based on the input of codes, tables, and user-written guidelines. Previous methods and benchmarks focus on understanding the codes or tables only, which can lead to descriptions that misalign with user intent.

collaborative writing that drafts purpose-driven descriptions from code and its execution results and lets authors guide, review, and refine the text. To enable this system, we first lay the groundwork: the dataset we construct and the model analysis we conduct here serve as preparatory components that pave the way for the downstream collaboration interface.

Documentation should capture both the code logic and the results it yields, because readers who encounter only one of these sources may misinterpret the analysis [167, 221]. Documentation is not a static deliverable; it changes with the analysis, captures intent and assumptions, and helps others interpret results later. This makes documentation a key stage between understanding the code and communicating what the results mean [26, 221]. Execution results can take many forms (*e.g.*, tables, charts, logs); we take tabular output as a common example and study how to document code together with such results. Yet documentation is often missing or under-specified around these outputs, because writing and maintaining descriptions interrupts exploratory work [160, 167, 189].

Automating these descriptions is tempting, but fully automatic generation does not match how documentation is used. High-quality descriptions need to be grounded in ev-

idence from both code and outputs, and they need to reflect the writer's purpose (e.g., a technical record versus a stakeholder update). Without this intent, there is no single correct description, and fluent text can still be misleading or factually wrong [148].

We therefore treat documentation as a human–AI collaboration problem. The system proposes drafts grounded in the available evidence (code and table outputs) and sensitive to user goals, while the author provides lightweight guidance, reviews what is written, and iteratively refines wording and emphasis.

Existing literature has explored techniques to generate descriptions for code snippets or table outputs (i.e., the rendered tables that appear when code is executed in a notebook, such as dataframes displayed by pandas) independently [8, 37, 40, 111, 123, 125, 146, 161, 191, 196, 228, 229], achieving satisfying performance. However, in real-world applications, documentations are always related to the corresponding code and table output, at the same time. For example, ground truth descriptions require summarization from the code and table output simultaneously, as shown in Section 4.1. This setting naturally calls for collaborative writing support, where systems propose drafts and users revise them based on context and goals.

Moreover, while the same code-table output pair could serve different purposes under different user requirements, existing work [8, 31, 37, 40, 61, 91, 98, 125, 134, 149, 161, 216, 222, 228, 229] only focuses on description generation for either code or table without considering various user requirements. As shown in Figure 4.1, under different user-written guidelines, the focus of different ground-truth descriptions could differ from each other.

To address the aforementioned issues and support the human–AI collaboration interface, we propose a challenging guideline-based text generation task for table and code description generation (**TCDG**). Given codes and output tables, the goal of **TCDG** is to produce concise descriptions under different user-written guidelines corresponding to different communicative purposes, as shown in Table 4.2.

As a preparatory step, we construct the first benchmark (**NBDESCRIB**) for TCDG that contains 3,924 processed code-table-description pairs extracted from highly-ranked notebooks from Kaggle competitions and identifies 15 guideline categories (details in Section4.3). Specifically, the raw Jupyter Notebook data with tables, code, and associated text from popular Kaggle competitions is crawled. However, the raw data cannot be directly used due to the large amount of noise [119, 131]. For example, *"I plan to refine the models by using more sophisticated machine learning techniques."* is about personal experiences and future plans; while such text may carry some contextual information, it does not describe the code or table output and therefore falls outside the scope of our generation task. On the other side, the user-written guidelines and ground-truth descriptions in the mark-

down cell generally contain multiple different purposes or facts on the tables. To reduce the noise in the raw data, we recruit annotators to first break down the markdown cells and make each piece of text only contain one purpose or fact. For each guideline category, we create the label as well as descriptions, and then curate the tables and filter out the noise text. Finally, the text descriptions in our data are natural, grounded in the source, and specifically targeted under different guidelines (Figure 4.1).

Next, we evaluate the performance of existing pretrained models. The ablation study shows that guidelines significantly affect the final performance at different levels, which demonstrates the validity of our task. Human evaluation further shows that these advanced models (*i.e.*, CodeT5, Llama, and GPT-4o) still struggle to produce well-grounded and purpose-aligned results, regardless of high-quality training data. Moreover, common error patterns involve misalignment with guidelines, incorrect variable values, omission of important code information, and reasoning errors.

Based on these preparatory results, we integrated our approach into the human–AI collaboration interface, an interactive system for collaborative writing, thereby completing the progression from dataset and model groundwork to the interactive tool. In a follow-up study, users found that the generated documentation reminded them of documenting code they might have overlooked and increased their satisfaction.

The main contributions of our work are:

- **Human–AI collaboration system:** an interactive system for collaborative writing that drafts purpose-driven descriptions and lets authors guide, review, and refine the text.
- **Preparatory work:** we formulate the **TCDG** task and collect the NBDESCRIB benchmark, which paves the way for the collaboration interface; experiments show that fine-tuned LMs (CodeT5) outperform powerful pretrained models (*e.g.*, GPT-4o and Llama), and ablation studies demonstrate that guidelines significantly enhance performance.
- Error analysis shows that LLMs fail to align with guidelines, understand variable values, and reason.

4.2 Related Work

Section 2.3.2 establishes the gap in joint code-table description generation: existing work focuses on code or tables independently, while in notebooks documentation typically relates to both. We address this gap by defining a table and code description generation (TCDG) task with explicit guideline categories for faithful text grounded in both sources.

Related table and code tasks. Table question answering [35, 139, 149, 199, 216] locates relevant cells and returns answers but does not produce explanatory descriptions for different data fact types. Table-based fact verification [32, 62] checks whether a statement is entailed or refuted by a table, but cannot generate descriptive statements. Template-based generation [5, 85] fills schemas with keywords but does not handle joint code-table input or varied user guidelines. Evaluation-oriented guidelines [44, 108] evaluate whether text communicates data meaning; they do not guide generation for multiple communicative purposes.

TCDG with guideline categories. Different from works that produce one text per code or table, our TCDG task allows one description to cover both code and its table output jointly, with multiple categories reflecting user needs (e.g., explaining data facts, summarizing code logic, highlighting trends). We construct a notebookTCDG dataset and provide category labels with descriptions as guidelines for generating faithful text in different directions.

4.3 TCDG - Task Description

In our task, the model is provided with a long text including a code cell and its table output, as well as the corresponding guideline category description. The guideline indicates the direction of the target description generation, and the specific category is described in Section 4.4.3. The model is asked to read the input and generate reasonable descriptions based on the given guideline, code, and table.

4.3.1 Input

The input to a text generation model consists of an input text and a target document:

(1) Codes and tables from the input texts are extracted from notebooks crawled from the Kaggle website. The code provides the necessary context to understand how the table output was generated. By analyzing the code, one can infer the logic and algorithms applied to the input data, which facilitates accurate interpretation of the table's contents.

(2) A **guideline category** serves as a guiding principle for generating the target description. Some descriptions focus on interpreting table content while code snippets also provide essential contextual information. At the same time, while describing the purpose

of the code, we also need tables to provide essential data-driven explanations. This setup mimics the real scenario.

4.3.2 Output

A text generation model is employed to generate descriptions based on the given guideline category, code, and table output. Table- and code-associated Markdown cells are the target documents that we collect since these cells are typically used to provide descriptive text for code and tables. Also, some Markdown cells can be used only for headings in the notebook. To exclude such Markdown cells, search for key characters like #, which generally refers to the titles.

4.4 NBDESCRIB

4.4.1 Data Collection

As we are focusing on the description generation in Jupyter Notebooks, we need to crawl a sufficient number of code-table-description triplets first. Publicly shared notebooks on GitHub are often ill-documented [166] and do not have many tables, thus are not suitable for this task.

On the other side, Kaggle allows community members to vote up and down on uploaded notebooks, and findings show that the highly-voted notebooks are of good quality and quantity for code documentation [125, 187]. Specifically, Wang et al. [187] found that top-voted Kaggle notebooks contain on average 45 markdown cells with detailed explanations, and that community voting serves as a quality filter because poorly documented notebooks receive fewer votes. Thus, we decided to utilize the top-voted and well-documented Kaggle notebooks. We crawled notebooks from seven popular competitions, *i.e.*, seven top popular Kaggle competitions - House Price Prediction, Titanic Survival Prediction, Predict Future Sales, Spaceship Titanic, U.S. Patent Phrase to Phrase Matching, JPX Tokyo Stock Exchange Prediction, and Ubiquant Market Prediction, and built around 4,000 code-table-description triplets. Links for these competitions can be found in Section B.3. To build this dataset, we also filtered out the description which is not in English. We checked the data policy of each of the competitions, and none of them have copyright issues. We also contacted Kaggle to make sure our data collection complies with the platform's policy.

	Overall	Train	Dev	Test
Code-Table-Description pairs	3,924	2,747	393	785
Code vocabulary size	3,497			
Table vocabulary size	16,424			
Description vocabulary size	4,481			
Avg. # token in Description	12.41	12.37	12.51	12.52
Max. # token in Description	66	57	46	66
Std. # token in Description	7.45	7.43	7.26	7.66
Avg. # token in code cell(s)	10.68	11.17	10.46	10.22
Max. # token in code cell(s)	310	310	131	131
Std. # token in code cell(s)	19.54	21.40	16.42	15.64
Avg. # token in table	13.85	13.92	12.84	14.11
Max. # token in table	272	272	97	261
Std. # token in table	17.27	16.47	11.97	21.68

Table 4.1: ***NBDESCRIB*** dataset statistics.

4.4.2 Data Preprocessing

We employed the following heuristics to collect codes, tables, and Markdown:

Cell Matching: We search for codes that produce tables in the notebooks and check if there is a corresponding Markdown cell describing the code and table above. The sentences are also split if there are multiple sentences in the corresponding Markdown cell. However, there are instances where the text content is inaccurate. For example, in texts labelled as value guidelines, the extracted value might be incorrect, or in texts describing Feature Engineer guidelines, the variable name extracted from the code might be misspelled. Thus, we label each sentence and let annotators rewrite it accordingly, since each sentence may have a different description angle. Details about how annotators code the sentence and reach an agreement are shown in Section 4.4.3. The specific guideline details will be described in the following.

Table Processing: Since the table in Jupyter Notebook is in HTML code, to transfer it into a table format, we use HTMLParser[1] to get the data value for each row, column,

[1]https://docs.python.org/3/library/html.parser.html

and their relationship based on the tags, such as <th>, <td>. We first drop their parent tags <table> to simplify the document format. Next, we remove the tags <td> and <th> from cells to extract variables and corresponding values from the HTML code. Then we concatenate variables and values with pipe("|") to generate table description. An example of table processing is shown in Appendix B.9.

Table Curation: If the description contains variable names in a table, the corresponding rows and columns containing those variables are extracted to create a new table. If no key variables are included, we keep the original tables. This process aims to minimize the inclusion of irrelevant information. Appendix B.9.3 provides an example of how to implement table curation.

4.4.3 Guideline Category Description

Three members of the research team conducted an iterative open-coding process to analyze the collected notebooks. Differing from Wang et al. [203], where their qualitative coding stopped at the tabular data level, our analysis goes deep to the granularity of the cell, where cells are used to explain beyond the adjacent code cell whose output is the table: we annotate these cells' purposes and types of content. Each annotator independently analyzed the same five notebooks to develop a codebook. After discussing and refining the codebook, they again went back to recode those five notebooks and achieved pairwise inter-rater reliability ranging from 0.81 to 0.93 across the three annotator pairs (Cohen's κ; the range reflects variation across the three possible pairings of annotators). To further determine the correctness of inter-annotator agreement, we let these three annotators analyze another undiscussed five notebooks and get pairwise inter-rater reliability ranging from 0.78 to 0.89 (Cohen's K) which is convincing to demonstrate the reliability of our codebooks. After getting a reliable agreement, the three coders divided and coded the remaining notebooks. In total, we identified fifteen guideline categories for the content of the markdown cells (Table 4.2, Appendix B.7 provides examples).

As shown in Table 4.2, eleven guideline categories mainly focus on the data facts of a table. It is worth noting while these guidelines focus more on the description of the table data, the code still provides contextual information to supplement their description, as shown in Figure 4.1. Our analysis revealed that markdown cells are mostly used to describe the specific attribute values from the table (Value, 7.29%). Second to the Value category, 6.55% markdown cells are used to specify the outliers from the table output (Outliers).

However, these eleven categories alone do not cover all the purposes that Jupyter Note-

book users have for markdown cells. This kind of markdown cells can also be used to mainly explain the beyond adjacent code cells. Even though they mainly focus on the code, a clear understanding of table data is also crucial for understanding the code logic.

Guideline	#	**Description**
Value	286 (7.29%)	Get the exact data attribute values for a set of criteria.
Difference	138 (4.02%)	A comparison between at least two distinct attributes within the target object, or a comparison between the target object and previously measured values.
Trend	31 (0.79%)	Indicates a general tendency over a period of time.
Proportion	120 (3.06%)	Measure the proportion of selected data attribute(s) within a specified set
Categorization	74 (1.89%)	Select the data attribute(s) that meet the condition.
Distribution	127 (3.20%)	Show the amount of shared value for the selected data attributes or present a breakdown of all data attributes.
Rank	73 (1.86%)	Sort data attributes by their values and display a breakdown of selected attributes.
Association	165 (4.21%)	Identify the useful relationship between two or more data attributes.
Extreme	227 (5.78%)	Identify the data cases that are the most extreme in relation to the data attributes or within a specific range
Outlier	257 (6.55%)	Determine whether there are unexpected data attributes or statistically significant outliers.
Aggregation	125 (3.19%)	Calculate the descriptive statistical indicators (e.g., average, sum, count, etc.) based on the data attributes.
Goal	771 (19.64%)	Express user's goal. To say what value or function they tend to use for the later research
Reason	276 (7.03%)	Express reason using the data from the table or explain the reasons why certain functions are used or why a task is performed.
Feature Engineer	393 (10.02%)	The process of selecting, transforming, extracting, combining, and manipulating raw data to generate the desired variables for analysis or predictive modeling.
Complementary Details	870 (22.17%)	Express additional contextual elements and supporting information to enhance understanding of the primary content

Table 4.2: We identify 15 guideline categories based on the types of descriptions in the Markdown cells which are below the code whose output is a table.

We found that some of these markdown cells describe the motivation from the code descriptive text(Goal, 19.64%), to explain the results or critical decisions (Reason, 7.03%), or to describe a combination of mathematical transformations from the code (Feature Engineer, 10.02%). We also found that some markdown cells aim to provide additional context to help with code and table understanding (Complementary Details, 22.17%).

4.4.4 Train / Dev / Test Splits

Overall, the dataset contains 2747 Code-Table-Description pairs in the training set, 393 pairs in the development set, and 785 pairs in the test set (see Table 4.1 for more statistics).

4.5 Experiments

Baselines. We utilized three types of models: a fine-tuned encoder-decoder-based CodeT5-Large, the popular decoder-only LLMs (an off-the-shelf GPT-4o, Llama3, CodeLlama3), and a fine-tuned GPT-3.0. Details are shown in Appendix B.4.

Models	R-1	R-2	R-L	BERTSc	Pyr	Coherence	Correctness	Consistency	Fluency	Orientation	Readability
Baselines											
CodeT5-Large	29.61	14.38	26.72	59.51	19.37	3.12	2.43	2.78	3.98	2.46	3.88
GPT-4o	26.06	3.57	26.18	53.62	18.29	3.23	2.38	2.61	4.13	2.42	4.15
GPT-3	19.25	4.42	20.72	51.26	13.37	2.64	2.19	2.16	2.89	2.08	2.86
Llama3	19.57	2.07	20.91	52.41	13.43	3.06	2.38	2.72	3.88	2.21	3.82
CodeLlama3	19.63	2.46	21.18	52.85	15.72	2.35	1.98	2.06	2.61	2.03	2.56
Ablation Study											
CodeT5-Large without table	22.40	9.54	20.49	55.91	18.52	2.55	2.06	2.17	3.74	2.02	3.53
CodeT5-Large without code	26.35	11.72	24.01	57.93	19.14	3.12	2.31	2.59	3.83	2.28	3.69
CodeT5-Large without guideline description	25.09	10.78	22.61	56.63	18.75	3.07	2.99	2.56	3.90	2.21	3.72
GPT-4o with chain of thoughts	24.91	3.82	25.42	51.25	16.91	3.22	2.87	2.69	3.84	2.48	3.86
GPT-4o with in-context learning	23.64	3.52	24.62	49.10	15.92	3.03	2.81	2.54	3.86	2.42	3.92

Table 4.3: ROUGE scores (R-1, R-2, R-L), BERTScore (BERTSc), Pyramid Evaluation (Pyr), and G-EVAL dimensions for the baselines, our model, and the ablation studies. Results show that this task is challenging though we use state-of-the-art text generation models.

4.5.1 Evaluation Metrics

We use the ROUGE scores (ROUGE-1, ROUGE-2, and ROUGE-L) [118], BERTScore [224], and G-EVAL [126] to evaluate our model's performance with regard to the ground-truth description content. Details are shown in Appendix B.8.

4.5.2 Results

The numbers in Table 4.3 show that this guideline-based text generation task is very challenging, while the fine-tuned CodeT5-Large obtained the best performance. It is interesting

to note that CodeLlama3 has bad performance in this task. The possible reason is that CodeLlama3 focuses mainly on code generation and is not suitable for description generation. As shown in Table 4.3, CodeT5-Large outperforms the GPT-4o, GPT-3, Llama3, and CodeLlama3 in this task. Additionally, we notice that the ROUGE-L of CodeT5-Large is above 25, and the ROUGE-2 is around 15, indicating that our dataset can produce more accurate and fluent text in response to different guidelines in this task. However, these metrics primarily check for semantic similarity. To further evaluate generated descriptions in diverse dimensions and achieve higher human correspondence, we conduct LLM-based evaluators G-EVAL [126] in six dimensions (Coherence, Correctness, Consistency, Fluency, Orientation, and Readability) on a 5-point Likert Scale. Details of G-EVAL are shown in Appendix B.8. We can find that CodeT5-Large outperforms the GPT-4o, GPT-3, Llama3, and CodeLlama3 in this task. It is interesting to note that existing LLM can generate fluent descriptions but struggle to produce coherent and consistent descriptions.

Ablation Study: To better understand the impact of each component on this new task, we perform ablation studies(Table 4.3) to evaluate how table, code, and guideline description contribute to the model performance separately. More concretely, we generate ablation models with the following settings: (1) without table, (2) without code, (3) without guideline description, (4) chain of thought prompting on GPT-4o, (5) in-context learning on GPT-4o.

Since CodeT5-Large performs best in the task, we use it as a backbone to test its performance without code, table, and guideline description. In general, all the elements contribute to the performance, and removing one element will lead to a significant performance drop.

Note that table content has a bigger effect on model performance compared to code. Code also influences performance by providing the necessary context to infer the logic, which aids in interpreting the table's content accurately. Guideline description can be seen as a synergy of tables that guide the generation system to generate desirable topics, and without it, the performance is slightly higher than one without any table content.

One intuitive method to enhance the reasoning ability of LLMs is Chain-of-Thoughts (CoT). Here we want to further answer this question: using CoT, can a large language model automatically find an optimum guideline and generate summaries better aligned with human interests? CoT is well known to work well for GPT models, so we experimented on GPT-4o with a CoT prompt containing both an example and middle steps of guessing a guideline (prompts shown in Appendix B.6). For a fair comparison, we also added the performance of in-context-learning for GPT-4o, by removing the provided guideline and directly providing the example (prompts shown in Appendix B.7). The result of CoT improved over in-context learning but is still inferior to the performance of the original

GPT-4o with ground-truth guidelines (except ROUGE-2).

Then we analyzed the match rate between guidelines generated through the CoT process and the ground truth. Results show that 72% of the guidelines did not match. Thus, although LLM can often generate readable and decent descriptions for code and table(see results in Table B.1 and Table 4.3), most of the generated descriptions are not as the users expected (see result in Orientation dimension in Table B.1). This demonstrates the necessity of guidelines. To fairly compare the generation models and standardize evaluation, we need to specify our guidelines for generating descriptions.

Pyramid Evaluation: We employ an automatic evaluation method based on pyramid evaluation (PyrEval) [56] to assess the faithfulness of generation. This metric, which correlates well with human evaluation, extracts and filters key phrases to preserve factual information, offering a more reliable metric to ROUGE. The results in Table 4.3 show consistent trends between PyrEval and ROUGE, validating the impact of table, code, and guideline descriptions in our ablation studies.

4.5.3 Human Evaluation

We conducted a human evaluation with 10 participants to assess the performance of various LLM models in generating code documentation. Participants rated 50 pairs of code and documentation across three dimensions: Correctness, Orientation, and Readability. The results showed that the Ground Truth outperformed other baselines. We defer the detailed results and analysis to Section B.1 to maintain the focus of this section on automatic evaluation; the human evaluation confirms the same ranking observed in automatic metrics.

4.6 Error Analysis

In this section, we analyze some common error cases in this guideline-based text generation task. Some examples can be found in the Section B.2.

(1) Variable values were generated and matched incorrectly. As shown in the example in Table 4.4, even though CodeT5-Large, GPT-4o, GPT-3, Llama3, and CodeLlama3 are capable of generating keywords such as "highest" based on the "Extreme" guideline, it remains difficult to produce accurate text content based on the variables in the table. For example, CodeT5-Large incorrectly predicted the oldest passenger as 80 years old. Table 4.4 also has this kind of error.

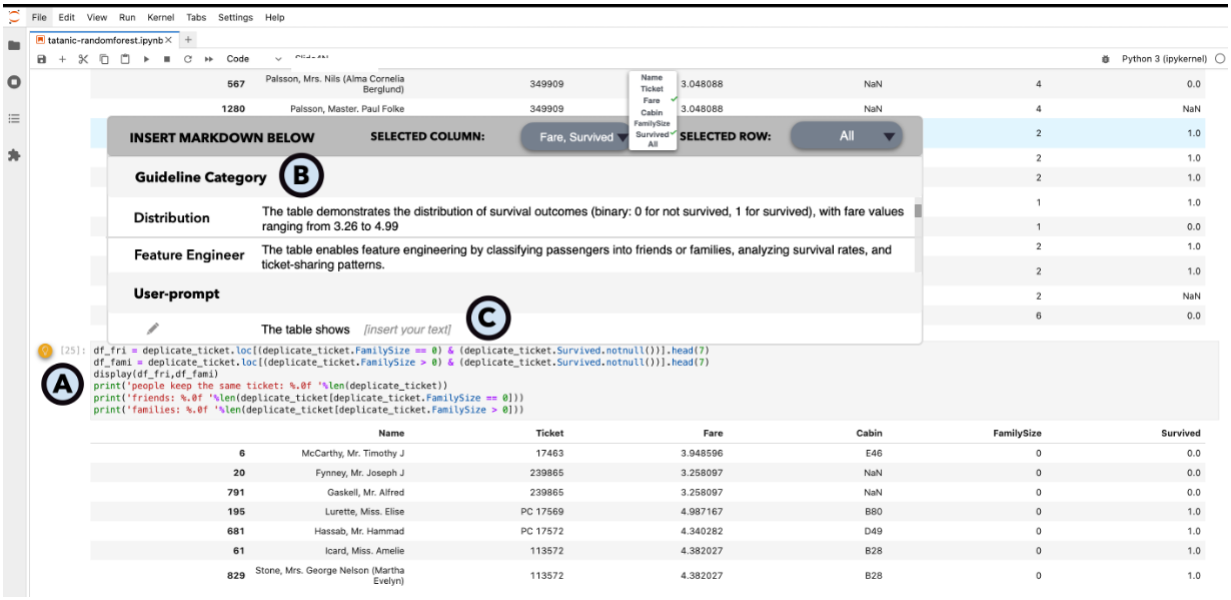


Figure 4.2: We implement a downstream application as a Jupyter Notebook plugin (A) to assist users in documentation writing for selected rows and columns from different guideline categories (B) and a user-prompt approach (C).

(2) The generated text focuses solely on the table and ignores important information in the code. In the example from Table B.6, ground truth is in the "Extreme" guideline and tends to convey that the first red wine has the highest pH value. However, the table does not have a related keyword "red wine." And CodeT5-Large failed to extract this information and also extracted the wrong value. GPT-4o cannot generate correct description either. Example from Table B.4 also has this kind of error.

(3) Generating incorrectly oriented text based on guidelines. For example, GPT-3, Llama3, and CodeLlama3 produce text related to "Difference", "Extreme", or "Distribution", but not "Trend" in the example from Table B.3. Another example in Table B.4, requires models generating text related to "Goal", but GPT-3 generates text related to "Association", describing the relationship between SibSP and Parch. Llama3 also generates text in "Feature Engineer" but not "Aggregation" in the example from Table B.2.

(4) Reasoning error. If operating under Aggregation guidelines, CodeT5-Large, GPT-4o, and GPT-3 may generate incorrect Aggregation data (count, mean, sum). In this example (Table B.2), GPT-3 can generate text such as this feature has many null values, but cannot obtain the count of null values.

We manually check 50 examples of CodeT5-Large, GPT-4o, GPT-3, Llama3, and CodeLlama3 models used in our user study and label the type of errors made. The most errors are made when they generate incorrectly oriented text (3rd type) **(54.1%)**. This is because the model tends to generate documents related to the best-trained guideline type in the dataset, such as "Association" or "Value". There are also two common errors made by generating documents with wrong values (1st type) and wrong reason (4th type) (**27%** respectively). Such errors are commonly made by generating "Value" or "Aggregation" type documents. There are also **13.5%** errors made by generating documents without considering the code. The dataset has numerous examples with insufficient code, leading the model to overlook some code instances.

These errors show that our task and dataset provide some challenges for existing foundation models. We firmly believe that researchers can enhance the existing foundation models in the future when they address the challenges. By building on our work and leveraging the valuable insights gained from it, they can push the boundaries even further, contributing to the continuous evolution of foundation models.

4.7 Downstream User Application

To demonstrate the application of our benchmark NBDESCRIB, we designed a Jupyter Notebook plugin to assist document writing in data science programming (as shown in Figure 4.2).

The plugin is triggered when detecting users focusing on a code cell (Figure 4.2.A). The plugin then reads the contents from the focused cell and its curated table output based on the selected columns and rows and sends the content to the backend. The backend server first generates documentation under different guideline categories using fine-tuned CodeT5-Large(Figure 4.2.B). In addition, we used prompts to nudge users to explain an output (Figure 4.2.C). If the user selects one of their preferred candidates, the chosen documentation will be inserted below the code cell.

Our plugin went through several rounds of pilot testing and iterative design. Participants found that it reminded them to document code they would have ignored, reducing the time spent developing documentation while actively exploring the data science task.

4.8 Conclusion

In this chapter, we formulated a new task, TCDG, that supports the *documentation* stage by automatically generating faithful descriptive text for code and table based on the given guidelines for a computational notebook. We collected a large amount of well-documented Jupyter Notebooks from Kaggle, resulting in a new benchmark dataset **NBDESCRIB**. From our analysis, our task imposed unique challenges to the current generation methods, including CodeT5 and LLMs. This dataset facilitated the task of creating practical slides for Jupyter notebooks [198] and enabled evaluations on faithful, high-fidelity, and factual generation.

One risk of AI-assisted documentation tools is that users may become overly reliant on generated text and lose the ability to write descriptions independently. This concern, sometimes called "deskilling," applies broadly to AI writing assistance [185]. In our setting, the interactive workflow mitigates this risk in two ways. First, the system presents candidates rather than inserting text automatically, so users must actively choose and edit. Second, the purpose-driven guidelines require users to think about the communicative goal before accepting a description. Nevertheless, longitudinal studies are needed to measure whether AI-assisted documentation affects users' independent writing skills over time.

One potential risk involves the substantial computational resources needed to run state-of-the-art language models. These resources consume significant amounts of energy, which not only raises the carbon footprint of such research but also leads to environmental degradation.

Guideline category description

Extreme: Identify the data cases that are the most extreme in relation to the data attributes or within a specific range

Code Cells

```
cols2 = X_test.columns.tolist() # List of column names
X_test = X_test[cols2] # Applying the new order
X_test
```

Table

Passenger	Sex	Age	Embarked	FamilySize
0	0	34.5	1	1
1	1	47.0	2	2
2	0	62.0	1	1
3	0	27.0	2	1
4	1	22.0	2	3

Description

Ground truth	The oldest passenger in X_test dataset is 62 years old
CodeT5-Large	Oldest person in the titanic was 80 years old and youngest person was less than one year
GPT-4o	It shows the initial rows of the table providing insights into the age distribution of passengers, with the label drawing attention to the oldest passenger
GPT-3	The oldest passenger is a man in his fifties
Llama3	The ages of passengers ranging from 27 to 62 years old with the youngest passenger being 27 years old
CodeLlama3	The average age of the passengers is 34.5 years old.

Table 4.4: An example of code-table cells and different models. Note that the table above is the original table, which will be processed by the table curation in Section 4.4.2 when generating the description. In this case, the curated table only keeps the passenger and age columns.

Chapter 5

Creating Presentation Slides from Code and Results

This chapter addresses **RQ(c)** from the introduction, focusing on the *presentation* stage of programming workflows. Building on the understanding and documentation support of earlier chapters, we recognized that well-documented notebooks still require presentation formats for communicating with stakeholders. How do AI-generated presentation artifacts support programmers in sharing their work with stakeholders and colleagues? We present Slide4N, a human–AI collaboration system for creating presentation slides directly from computational notebooks. The system integrates model-based slide generation into a JupyterLab plugin that keeps authors in control of scope, structure, and edits, reducing manual assembly work while preserving author control.

5.1 Motivation

This chapter addresses the *presentation* stage of the programming workflow introduced earlier (Figure 1.1). We develop an interactive human–AI collaboration tool that enhances this stage by assisting data scientists in creating presentation slides from computational notebooks.

Computational notebooks (*e.g.*, Jupyter Notebook [82], JupyterLab [83]) have become a dominant environment for data exploration and analysis in data science [95, 167]. They combine code, notes, and outputs (*e.g.*, tables and plots) within a single document, supporting iterative work that blends computation with explanation and interpretation [90,

100, 152]. In almost every industry, with the growing complexity of the work, multiple collaborators are usually involved in the same analysis [135, 221]. The communication between technical members (*e.g.*, model builders) and non-technical members (*e.g.*, managers, domain experts, stakeholders) is critical and frequent [41, 101, 194]. Despite the sharing capability of notebooks, directly using them to present findings is challenging [27]. For example, a typical notebook tends to be long, poorly formatted, and interspersed with interim notes and findings [67], which is not intended for presentation [167], and can be difficult to understand for other members of a team, especially non-technical ones. This reflects a challenge at the *presentation* stage: turning computational artifacts into shareable knowledge with a clear narrative and emphasis that matches a specific audience. Doing this work by hand is slow and error-prone when content must be transferred and kept consistent across tools [27, 153]. At the same time, fully automatic slide generation cannot decide what matters, how to frame the story, or how to balance detail and clarity without access to human judgment. These constraints point to Human–AI collaboration in slide authoring: the system can draft structure and content, while data scientists decide scope, adjust narrative, and refine details.

Data scientists are then forced to use other presentation tools (*e.g.*, Google Slides, Microsoft PowerPoint) to create slides for the sake of communication [27]. This causes a phenomenon, we call, *analysis presentation divide*. Much tedious and repetitive work needs to be done to transfer contents between the computational notebooks and the slides [153], since they are fundamentally two different mediums. A few steps [53, 230] are required for data scientists to create a high-quality presentation from their analysis. When creating a slide around a topic in mind, relevant cells first need to be *located* and essential information needs to be *distilled* from the complex, messy, and loosely-connected code [101, 167]. Moreover, all kinds of information such as text, plots, tables, etc. need to be *arranged* in a suitable layout to provide a narrative. Lastly, the slide needs to be *refined* with customization to accommodate different domains and audiences [21, 178]. Data scientists need to frequently switch between the notebook environment and the presentation tool, with a lot of copying and pasting, which is error-prone. What is worse, any updates on the analysis (*e.g.*, a change of resulting plots) would trigger the remaking of relevant slides.

There have been some attempts to address the above challenges. ToonNote [88] allows users to craft a data comic from a notebook by putting visualizations and annotations in a sequence of frames; however, it still requires a lot of manual operations and the results are quite different from a slide deck. Themisto [190] automatically generates short Markdown annotations (*i.e.*, titles) for code snippets in a notebook, but not in a ready-to-use presentation format. Some other researchers have attempted to automatically create slides from text articles [176] and scientific documents [53]. However, these methods cannot

be directly applied to notebooks because of the complexity of their format which includes code, text, tables, plots, etc. The closest work to ours may be NB2Slides [230] that uses a template-based method to generate an entire slide deck from a notebook. Nonetheless, it requires very restricted input notebooks that need to have few excess code cells, high-quality documentation, and a complete data science workflow, because of the template-based approach; these requirements are difficult to achieve in data scientists' day-to-day activities.

In this chapter, we take an interactive, collaborative approach to augment the slide creation workflow from a notebook. The system proposes candidate slide structures and phrasing, while users iteratively steer, edit, and validate the resulting narrative. We design and develop Slide4N (Figure 5.1), an interactive assistant for data scientists to create slides from computational notebooks, which is built within JupyterLab to provide a seamless user experience. To create a slide, a data scientist just *locates* related cells (including code and Markdown cells), and then Slide4N automatically *distills* key information to generate useful title and bullet points on the slide, using advanced natural language processing (NLP) techniques. Moreover, Slide4N algorithmically *arranges* the contents in groups and further displays them with appropriate layouts. It also provides a set of easy-to-learn interactions to allow for *refining* the generated slide with user customization. High-level human input to the Slide4N automation is further allowed for refinement and customization, such as showing more or fewer details on slides. A data scientist can iteratively select cells and create slides one by one to form a presentation. The slides are fully linked with the computational notebook, allowing for effortless synchronization between the analysis and the presentation. Our human-AI collaborative approach is versatile and dynamic, which does not require an entire analytical workflow completed in the notebook, or many restrictions on the quality of code and documentation. The system automates the tedious parts of slide creation, leverages suitable user inputs, and produces results in a commonly-used presentation style.

We evaluate Slide4N by conducting a two-part study, where in Part 1 participants ($N = 12$) used a provided Kaggle notebook to create presentation slides, and in Part 2 participants ($N = 6$) used their own notebooks to do the same task. We also invited the six participants in Part 2 and two other experts to rate all the created slides on five aspects (*e.g.*, satisfaction, clarity, organization, etc.), and the results demonstrate that Slide4N could help participants generate high-quality slides. Moreover, questionnaire responses and qualitative feedback from the participants indicate the effectiveness and usefulness of the tool. In general, participants appreciated such a human-AI collaborative approach compared to fully-manual or fully-automatic methods. We also distill some design implications from our study to inform the future design of such tools.

In summary, our contributions in this chapter include:

- An analytical pipeline that enables interactive and iterative slides creation from computational notebooks powered by recent NLP techniques;
- An interactive human–AI collaborative tool, Slide4N, built within JupyterLab, that assists data scientists in creating presentation slides with a seamless experience; and
- An evaluation of Slide4N that illustrates the value of our approach and inspires the design of similar tools.

5.2 Related Work

Chapter 2 identifies the notebook-presentation divide and reviews RISE, NB2Slides, and tools for notebook readability. We focus here on additional tools and on how slide content is generated.

5.2.1 Presentation Support for Computational Notebooks

ToonNote [88] allows users to curate data comics from notebook outputs and Markdown cells to improve comprehension, but requires many manual operations; Slide4N automates much of this via NLP and analytical techniques. As Section 2.3.3 discusses, RISE transforms cells to slides as-is without distilling key points, and NB2Slides uses a template-based approach by retrieving content via NBSearch [112] that requires few excess cells, high-quality documentation, and a complete workflow—difficult to achieve in exploratory notebooks. NB2Slides also lacks user interaction support and automatically groups cells, making it hard for users to modify slides. Slide4N loosens these restrictions with a human-AI collaborative approach: data scientists select cells of interest, refine slides incrementally, and control topic, auto-merge options, and detail levels while keeping notebook and slide content synchronized.

5.2.2 Slides and Code Documentation Generation

Document-to-slide systems generate slides from papers or documents, not notebooks. Fu *et al.* [53] use a hierarchical seq2seq model with paraphrasing and layout prediction. Sun *et al.* [176] present a query-based method where users enter a title and the system selects sentences via Dense Vector IR. Hu *et al.* [72] select important sentences using Support

Vector Regression. These approaches target scientific documents, which differ from notebooks that interleave code, Markdown cells, and outputs. For slide content from notebook code, we draw on code documentation models. HAConvGNN [124] generates high-level documentation for multiple notebook cells and inspires our title generation. CodeT5 [201] provides identifier-aware code understanding. Slide4N leverages CodeT5 to generate bullet points from user-selected code cells, and HAConvGNN plus a fine-tuned T5-based model for title candidates. Themisto [197] supports interactive documentation during coding but targets technical users, not slide presentation.

5.2.3 AI-Infused Systems for Data Science

Data science is a labor-intensive field consisting of many investigative tasks such as exploratory data analysis, model development, and hypothesis verification. To improve the efficiency of data scientists, researchers developed many AutoML techniques to automatically process data, train models, perform feature addition, and other data science tasks [54]. In addition, many IT companies (*e.g.*, H2O [106], Azure Machine Learning Studio [15], Google Cloud AutoML [58]) developed some AutoML applications to allow both technical and non-technical users to perform general data science tasks and get insights in a faster and easier way.

There are also some interactive tools that use AutoML to help users do data exploration [120, 184]. Tsiakmaki *et al.* [184] utilized AutoML to predict students' learning outcomes based on their participation in online learning platforms. Liu *et al.* [120] incorporated AutoML algorithms to do hyper-parameter searches for various kinds of genomic profiling data. However, most of the AutoML techniques only focus on the main data science tasks such as data processing, feature engineering, model selection, and model building, overlooking the presentation support in the whole workflow, which is our focus in this chapter. Further, while fully automatic systems are powerful, there is still much deficiency due to errors and sub-optimal performance of the models. Thus, we leverage a human-AI collaborative approach to automate parts of the slide generation process while allowing users to intervene for achieving better results overall.

Moreover, some interactive AI-Infused tools have been created with friendly integration with the computational notebook setting. Wrex [42] is a Jupyter Notebook extension that supports data transformations with a programming-by-example method. B2 [211] treats data queries as a shared representation between the code and interactive visualizations allowing users to easily move from code to visualization and vice-versa. BURRITO [60] automatically infers a researcher's computational activities by capturing and displaying code

outputs with notes. ATENA [13] is a system that takes an input dataset and automatically generates a compelling exploration session, displayed in an EDA notebook. EDAssistant [113] develops an extension to support exploratory data analysis by recommending useful APIs and searching code in the Jupyter Notebook setting. NBSearch [112] provides a semantic code query to help data scientists find relevant cells in a large notebook corpus. This idea inspires our design of Slide4N on auto-merging relevant cells into groups to offer a more concise slide generation and also render the slide layout according to the cell groups. In this work, we aim to employ a human-AI collaborative approach for slide creation, instead of a fully automated system. This way, it provides much flexibility by combining the power of machine automation and human creativity.

5.3 Slide4N Design

In this section, we first describe the design goals that guide the development of Slide4N, then provide an overview of the system, and finally demonstrate the usage of Slide4N with a simple scenario.

5.3.1 Design Goals

We distill the following design goals to inform our development of Slide4N. These design goals are mainly extracted from the literature and the common challenges of data scientists in making presentation slides based on their notebooks.

G1: Link the notebook and presentation slides. Computational notebooks tend to be messy, poorly formatted, and less documented, which are hard for other members of a team to understand or even one's future self [67, 190]. To facilitate communication, data scientists are forced to transfer contents from the notebooks to the slides using other presentation tools (*e.g.*, Microsoft PowerPoint) [27, 153]. But the exploratory and analytical data science work often requires rapid iterations, which results in frequent updates of presentation slides based on code in notebooks. Thus, the system should interactively link notebook cells and slides to maintain the association and provenance between them [68, 193], which can also provide technical context to data scientists when making slides.

G2: Help locate relevant cells in the notebook for creating slides. While the code in traditional software is linear, the code cells in notebooks are relatively independent; they can be executed in arbitrary order and executed multiple times. This results in loosely connected codes in notebooks [67, 175, 207], so the cells before or after one cell may not

be necessarily connected. On the contrary, when presenting the analysis in the notebook, the contents need to be linear, forming a narrative to allow a better comprehension. Each individual slide is usually based on a set of tightly connected cells, and thus there is a gap between the cells in the notebook and the information needed to create a slide. Therefore, the system should provide an easy way for users to locate these cells, so that they can gain sufficient information to create a slide.

G3: Assist with distilling key information from code and Markdown cells. Readers seldom have an interest in reading code when trying to understand the notebook at a high level, even for the closest collaborators [165]. Thus, a presentation slide typically does not contain any raw code; instead, it should display contents that are more human-readable, self-explanatory, and concise, such as in titles, bullet points, and pictures/tables. However, it is usually time-consuming to manually digest and summarize the key information from a lengthy, messy, and poorly-documented notebook. Thus, the system should enhance this information extraction and summarization process with automation.

G4: Arrange slide contents in a logical and coherent narrative flow manner. Unlike the notebook code and markdown cells that are loosely connected, the contents (*e.g.*, headlines, points, and pictures) on a slide are often tightly interrelated [11, 204]. Properly organizing them in groups and displaying them with an appropriate layout can help structure the delivered information, create a narrative flow, and aid understanding [204]. Even with well-curated information, manual organization and layout of the slide contents are tedious and can lead to inconsistent slide styles that raise the cognitive burden on readers. Most presentation tools (*e.g.*, Microsoft PowerPoint) provide several layout templates to mitigate this issue. However, data scientists still need to manually place the contents to certain layout components. Therefore, the system should take the relationship of the presenting information into account and automatically arrange the contents logically and elegantly.

G5: Support necessary human intervention for slide generation. Automating the slide generation is convenient, but the quality of the generated slides (outputs) cannot be guaranteed due to known and unknown limitations of machine learning models [19, 232]. Also, for fully-automated systems, there usually exists little control for users during the entire generation process, which could decrease the overall efficiency when the outputs do not meet the user's expectation [53, 176, 230]. A better solution is to place humans in the loop, enable human-AI collaboration, and balance the trade-off between automation and human agency [69, 137]. Therefore, the system should provide intuitive user interactions to allow for the right amount of control over the slide generation model, such as selecting the input, tweaking high-level model parameters, and adjusting generated slides.

5.3.2 System Overview

Based on the aforementioned design goals, we developed Slide4N, an interactive and intelligent system that supports data scientists in creating slides from computational notebooks with human-AI collaboration. An overview of the system architecture is shown in Figure 5.1, which includes two parts: an NLP-enhanced back-end and a front-end user interface, working together to fill in the gap between analysis and presentation needs in data science. Details of each component will be introduced in the following sections. Here, we provide a high-level description of the system.

In particular, based on the user-selected cell(s), the back-end computes cell similarity to help data scientists locate relevant cells (G2). It then analyzes the contents of the selected cells and extracts key information to generate meaningful slide titles, bullet points, and other contents such as plots (G3). This can provide a good "first cut" for data scientists to describe the work in selected cells. The back-end further organizes these contents based on cell relevance and suggests an appropriate layout for rendering the slide (G4).

With the support of back-end, the front-end user interface is implemented as a JupyterLab extension to provide a seamless experience for data scientists to create slides from computational notebooks. As shown in Figure 5.1, Slide4N can be used together with the main JupyterLab notebook window side-by-side, which consists of four interactively-linked components: (a) a *Code* that visually summarizes the code and markdown cells in a notebook and their relationships (G1), (b) a *Control* that allows users to provide high-level guidance to the slide generation (G5), (c) a *Content* that displays the outline of the working slide deck, and (d) a *Slide* that supports interactive customization and refinement of the generated slides (G5).

5.3.3 Usage Scenario

Here, we use a simple scenario to demonstrate an interactive iterative workflow of creating slides from a computational notebook with Slide4N. Suppose that Crystal is a data scientist who works in a real-estate startup. She uses JupyterLab to explore and analyze a dataset of housing prices in Toronto, Canada. She has finished an initial analysis and drawn some conclusions, now she needs to present her results to other stakeholders consisting of a mixture of technical and non-technical audiences such as her fellow data scientists, program managers, and business people. Moreover, she needs to produce a deck of slides in a relatively short amount of time as required by her manager.

Crystal thus opens her notebook in JupyterLab and clicks the "Slide4N" button on the

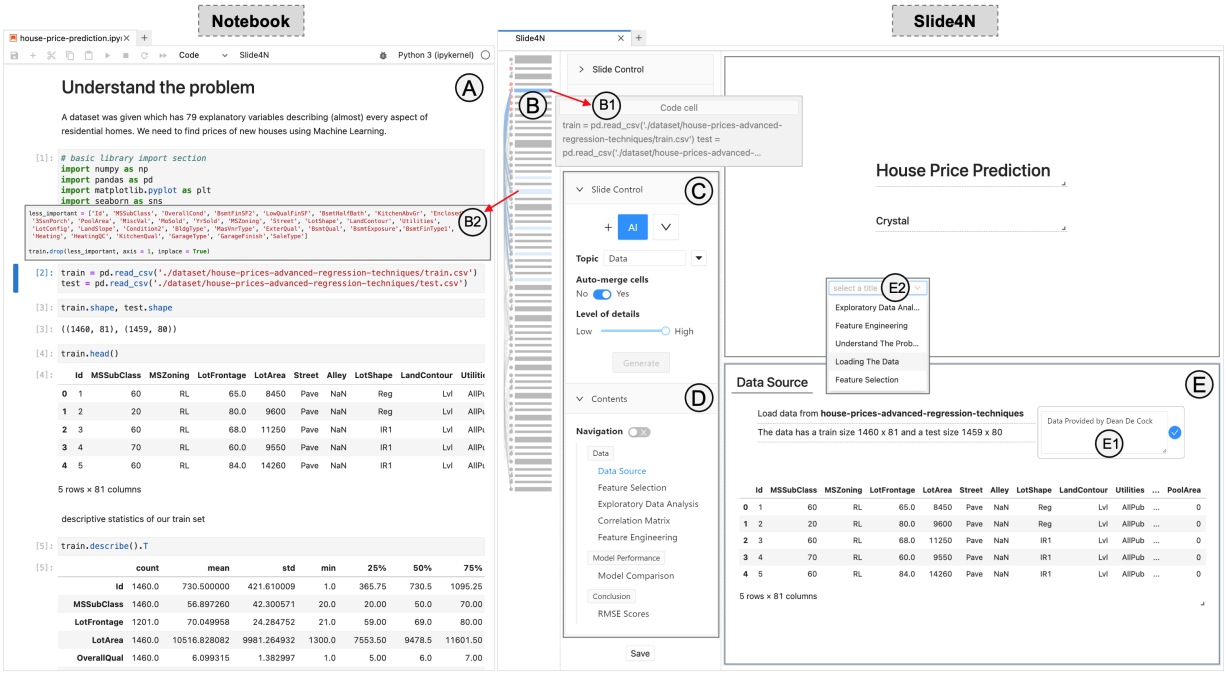


Figure 5.1: Slide4N is an interactive AI assistant for data scientists to create slides from computational notebooks, which can be used side-by-side with JupyterLab (A). The front-end user interface of Slide4N consists of four interactively linked components: a *Code* (B) that visually summarizes all the cells in a notebook and their relationships, a *Control* (C) that allows users to provide high-level guidance to the slide generation, a *Content* (D) that displays the outline of the working slide deck, and a *Slide* (E) that supports interactive customization and refinement of the generated slides.

toolbar to launch Slide4N. She then puts the two windows side by side to start to create slides based on her notebook (Figure 5.1A). First, she wants to create a slide to introduce the dataset used in the analysis, which is beneficial for people who are not familiar with her project. She clicks the "AI" button on *Control* (Figure 5.1C) that creates an empty slide on *Slide* (Figure 5.1E) with some necessary tips on it: select relevant cells, configure the parameters, and refine the generated slide. Following the tips, she goes to the *Code* (Figure 5.1B) to browse the entire notebook. When hovering over the rectangles, which represent code and markdown cells, a tooltip is shown to display the basic information about the corresponding cell (Figure 5.1B1). She clicks a cell that loads data which she thinks is quite relevant to the slide in her mind, and then a set of curves appear on *Code* to indicate the relevant cells to the selected one based on shared variables. These relevant cells

are also color-coded with shades of blue to indicate the extent of relevance. Meanwhile, Slide4N automatically scrolls down the notebook window to the actual cell, so that Crystal can view the detailed code. She then finds some cells nearby that are related to the topic of the slide to create, and selects them.

After, Crystal moves to the *Control* (Figure 5.1C) to configure the automatic slide generation at a high level. The selected cells are all about data, so she sets the "Topic" to Data. Considering that many cells are selected and they are messy, she also activates "Auto-merge cells" which can help her merge relevant cells. Finally, she sets the "Level of details" to the highest because she wants a more detailed version of the slide contents, which would be easier for others to understand. After this simple configuration, she clicks "Generate" and within seconds, a slide with generated title, bullet points, and plots is rendered on *Slide* (Figure 5.1E). These contents are also nicely laid out.

However, Crystal thinks the generated title doesn't fit her style, so she rephrases it. She is also not satisfied with the first generated bullet point, so she slightly changes it. In addition, she adds another bullet point (Figure 5.1E1) to describe the source of the data, which is not exhibited in any cell of her notebook, but now she thinks it is nice to cite the source for a presentation. She further rearranges the slide contents based on the originally suggested layout by Slide4N. Crystal is now satisfied with the slide and moves on to creating other slides in a similar manner. The system is versatile enough to analyze the relationships among the selected cells to extract key information and suggest different slide layouts based on it. With the assistance of Slide4N, Crystal quickly creates a slide deck for her presentation and experts it for the meeting.

5.4 Slide4N

The scenario above is made possible in Slide4N through a series of operations (Figure 5.1), including locate, distill, arrange, and refine, which are detailed below.

5.4.1 Locate: Finding Relevant Cells

The Slide4N back-end maintains a cell relevance graph which is built upon the most up-to-date cell information captured by the front-end (G2). From each code cell, we extract structural and content level granularity. The usage of the abstract syntax tree provides levels of context to the inherent nature of programming languages. Our algorithm characterizes the semantic coherence between cells by aggregating the percentage of overlapping

attributes and identifiers between code cells. For pairwise relevance calculation, we define the relevance score as:

$$S = 1 - (1 - \frac{x}{a}) \cdot (1 - \frac{x}{b}) \tag{5.1}$$

where a denotes the number of variables in the source code, b denotes the number of variables in the destination code cells, and x denotes the shared number of variables between source and destination code cells. The score ranges from 0 to 1 where its maximum is achieved when variables from either of the code cell are a subset of the other one, and the minimum is achieved when variables from both cells are strictly exclusive to each other. Following the formula, the back-end composes a weighted, undirected, and complete graph of the notebook. The data is sent to the front-end to render in the *Code* that indicates all the cell relevance. For example, in Figure 5.1B, rectangles with shades of blue indicate relevant cells. Figure 5.1B1 is selected by the user (the source code), B2 is a relevant cell (the destination code), and they share the "train" variable.

The *Code* (Figure 5.1B) concisely displays the cells of the notebook and their states, linking the notebook and Slide4N (G1, G2), allowing a user to select cells used for later slide generation and displaying the cells associated with the slide. Each cell of the notebook is visualized as a rectangle with equal width. The more content the cell contains the higher the rectangle is, and the rectangles are placed from top to bottom to align with the order of the cells in the notebook. The dot on the left of each rectangle reflects the state of a cell with colors, including default (gray), focused (blue), and selected (pink); and a red border indicates the cell being updated. For markdown cells, there is a # below the dot, so that users can quickly distinguish different types of cells.

Several basic interactions are provided to facilitate the navigation of the notebook. When the user hovers over a cell, a tooltip pops up and gives a quick overview of the cell content, including the cell type, the first three rows of code, and the output plots or tables (if any). Also, the related cells sharing variables and functions are connected by blue arcs. When the user clicks on a cell, all the related cells are highlighted in blue (if any). A darker shade means greater relevance, and the current cell is always the darkest. At the same time, the notebook window jumps to the corresponding actual cell.

5.4.2 Distill: Generating Contents from Cells

Based on the selected cells, the back-end of Slide4N generates two types of content for each slide: titles and a set of bullet points(G3).

Title Generation. In total, four or five slide title candidates are recommended to the users using various methods, because we want to provide users with different detail levels

Topic (tags)	Candidate slide titles
Introduction	Purpose, Workflow
Data	Data Loading, Data Preparation, Exploratory Data Analysis, Data Preprocessing, Feature Engineering
Model	Model Building, Train-Test Splitting, Model Training, Model Parameter Tuning, Model Validation
Model Performance	Performance, Model Interpretation
Conclusion	Limitations & Future Work, Suggestions

Table 5.1: The association between distinct topics (tags) and slide titles that Slide4N uses to generate title candidates which are further selected using SequenceMatcher.

of title recommendations and ensure the usability of the recommended titles. For example, Figure 5.1E2 shows the title candidates in a list, where the first two, third, and last two candidates are from the first, second and last methods, respectively. The fourth candidate is usually more accurate so it's selected by default.

First, the Slide4N back-end uses five topics (tags)—Introduction, Data, Model, Model Performance, and Conclusion—proposed in NB2Slide to generate two candidates, because these topics reflect the main stages of a data science analysis workflow [230] that can be the seeds of suitable summary titles. But compared to NB2Slide, we provide different titles for each topic after conducting a think-aloud formative design session with three data engineers, as shown in Table 5.1. Based on the user-specified topic in *Control* (Figure 5.1C "Topic"), two candidates are selected by using the SequenceMatcher [1] from the Difflib library to find the text correlation between the title and the selected cells.

The third candidate title is generated from markdown cells. If there are markdown cells selected by the user and they contain hashtags (indicating user-documented titles), the system extracts this hash-tagged sentence and processes it for generating the slide title candidate. However, if no markdown cell is selected, the title is not generated.

The last two candidate titles are generated by two neural-network models. We employ two neural-network models, each generating a candidate title that more specifically summarizes the selected code. One model is HAConvGNN [124, 195] and the other is the T5-Base model [157] fine-tuned by our own collected notebook database. Even though high-level documentation might be too detailed sometimes for the title, users still need this kind of

[1] https://docs.python.org/3/library/difflib.html

title for an in-depth presentation. Inspired by Mastropaolo *et al.* [127] who demonstrated a good performance after fine-tuning a T5-based model in code-related tasks, we also fine-tuned a T5-base model to generate other candidate titles. The reason for choosing two models is that we find sometimes the recommendation of one model is not satisfactory, and the two models can play complementary roles.

Bullet Points Generation. Besides the slide title, the Slide4N back-end generates one bullet point for each selected cell by using the codeT5 [201] model, which is the state-of-the-art code documentation generation model. We directly use the model for bullet point generation without fine-tuning since its performance is already good enough(G3).

However, in some cases, the generated bullet points may be too detailed or too high-level based on the number of selected code cells. Slide4N provides two user interactions with *Control* (Figure 5.1C) that allow users to customize the complexity of the generated bullet points (G5). The first is an "Auto-merge cells" switch, and when turned on, the selected code cells are grouped based on their relevance scores (see Section 5.4.3 for details) and concatenated before being sent to the model. The second is a "Level of details" slider (Figure 5.1C) that controls the detail level of the generated bullet points and offers three different levels. According to the user's choice, the corresponding hyper-parameters of the model are adjusted to generate different bullet points. The above two interactions are also useful for different presentation scenarios. For example, if the main audience includes non-technical stakeholders who need a more concise description, the "Level of details" can be set to the lowest and the "Auto-merge cells" can be turned on; and when most of the audiences are experienced engineers with a deep understanding of related technologies, the "Level of details" can be set to the highest and the "Auto-merge cells" can be off.

5.4.3 Arrange: Organizing and Laying out Contents

In addition to the generated slide title and bullet points, the system captures the outputs (*e.g.*, plots and tables, if any) of the selected code cells and renders these contents on a slide (G4). The slide title is obvious, however, other contents need a meaningful layout. To do so, the Slide4N back-end goes through the following four-step process, inspired by Zhao *et al.*'s [227] data comic layout generation method based on "narrative backbones" (*i.e.*, a tree structure logically connecting the layout elements). Essentially, we construct a two-layer narrative backbone for arranging various slide contents in our case (Figure 5.2).

First, based on the calculated relevance scores for selected cells (see Section 5.4.1), Slide4N composes a complete weighted graph (Figure 5.2-1), where each node in the graph represents a cell and each edge represents the relationship between the cells. The relevance

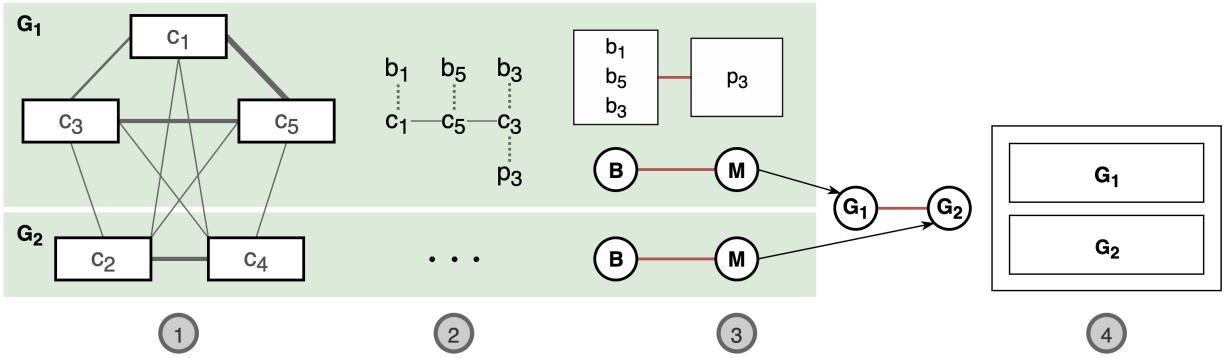


Figure 5.2: The four-step process to arrange contents on a slide (*e.g.*, bullet points, plots, and tables): (1) separate user-selected cells into groups based on the relevance scores; (2) arrange bullet points and cell outputs based on the logical order of cells; (3) align text and plots by linking them (red lines) and construct the backbones within and between groups; (4) map the two-layer backbones to the layout design patterns (see Figure 5.3 for details).

score between two cells is defined as the edge weight. After constructing the graph of selected cells, we apply hierarchical agglomerative clustering (HAC) [64] to separate the cells into groups (subgraphs). For example, in Figure 5.2-1, there are two groups, where G_1 includes cells c_1, c_3, c_5, and G_2 includes cells c_2, c_4.

Then, for each subgraph, we compute a maximum spanning tree to find the subsets of edges that connect all the cells with the biggest total relevance score. This tree represents the structure of the most relevant connections for the selected cells. Based on the execution order and the relevance score of the code cells, we conduct an internal walk-through from the root node of the tree to a leaf node; this order is likely to follow the logical order in the narration. The execution order is used to select the root node and determine the order of multiple cells with the same relevance score. For example, in Figure 5.2-2, c_1 is the root node and the tree is from left to right. We use this walk-through to arrange bullet points generated by different cells, and cell outputs (*e.g.*, plots and tables, if any) in a linear form, respectively. As shown in the upper part of Figure 5.2-3, the bullet points (b_1, b_3, b_5) in the group G_1 are ordered (from top to bottom) and placed together (denoted by B), and the output of c_3, a picture p_3 (denoted by M) is also included.

Next, we form an intra-group backbone by linking M and B elements from the same cell (*e.g.*, the red lines in Figure 5.2-3). This is because the generated bullet points and cell outputs are naturally bonded to the cell (indicated by the dashed lines in Figure 5.2-2). Similarly, we build a weighted complete graph using the root cell from each group to

linearly order different groups, thus forming an intergroup backbone as illustrated on the right of Figure 5.2-3. Based on the two-layer (intra-group and inter-group) backbones, we generate the final layout (Figure 5.2-4). For a slide that has four or fewer elements (*i.e.*, nodes in the backbone), there are seven different structures to reflect the logical association between them (Figure 5.3). This covers most of the cases after grouping the cells, however, there are occasions when a slide contains more than four elements. We then place the elements linearly from left to right, for example, bullet points on the left and plots/tables on the right.

Last, the system converts the narrative backbones to the layout design patterns, which is inspired by Bach *et al.*'s data comic layouts [11]. We use a rule-based approach to map the two-layer narrative backbone to the layout patterns in the presentation space, as detailed in Figure 5.3. One narrative backbone might be reasonably matched with multiple layouts. Currently, however, there are no agreed rules or principles for automated matching. Thus, based on Wang *et al.*'s survey for layouts in infographics [202], we adopt frequently-used layouts such as Tiled (40.4%) and Parallel (28.6%). We also consider the space efficiency of the layouts, such as balancing the size of points and plots/tables and increasing the data-ink ratio. For example, in Figure 5.6b, the left is the user-selected notebook cells, and the top right is the generated slide with contents properly organized. The selected cells first sort the prediction models, and then visualize their RMSE scores. Accordingly, the generated bullet points and associated cell outputs are separated into two groups. "Parallel" and "Linear/Parallel" are used for the inter- and intra-group backbones, respectively. White gaps are also used to distinguish different groups.

5.4.4 Refine: Customizing Generated Slides

From the generated slide, which can be a "first cut," the user can make further refinements directly on the *Slide* (G5). It renders the slide contents and layout configuration that are obtained from the back-end.

Overall, each slide in the *Slide* (Figure 5.1E) is divided into title area (top) and content area (below), which uses a grid-based approach to place bullet points, plots, tables, etc., all of which can be individually dragged and dropped to adjust the layout. The design of *Slide* draws on commercial presentation tools like Microsoft PowerPoint and provides a set of accessible interactions for users to refine and customize the generated slides. For example, the user can add new bullet points, select generated bullet points, and modify them using markdown-like grammar. The user can also insert, resize, and remove plots or tables, as well as order, delete, and restore slides, etc. In cases where the user wants to create a

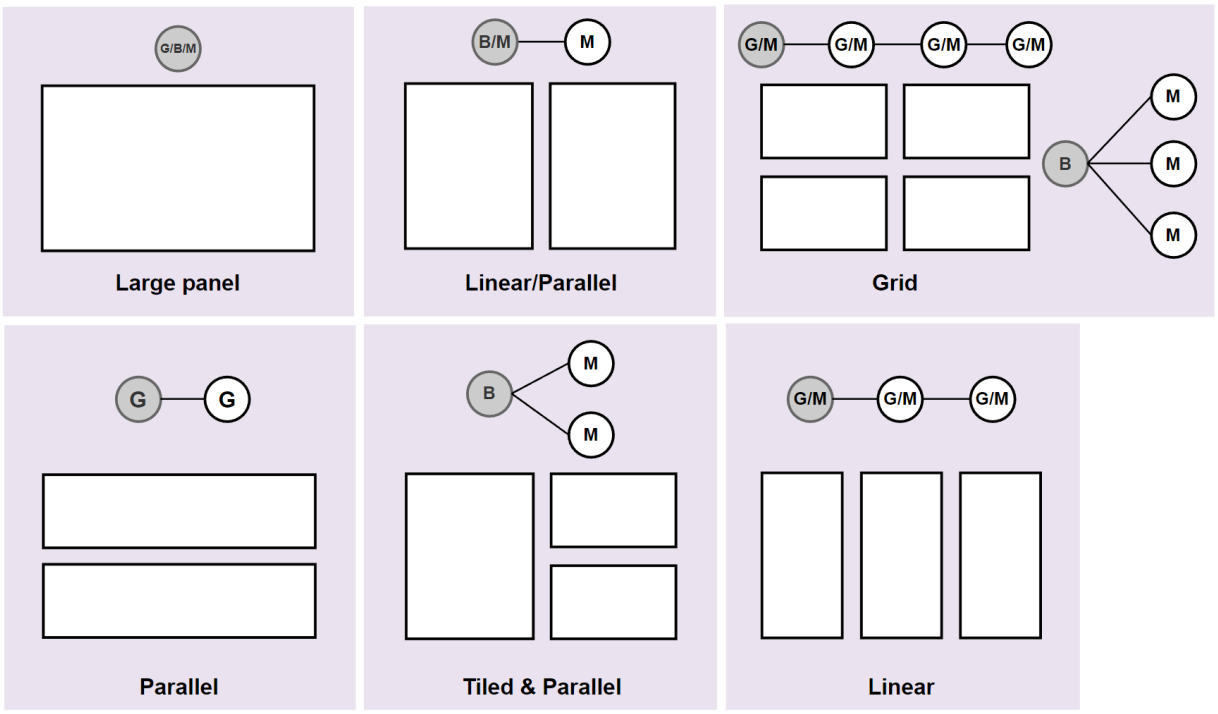


Figure 5.3: The mapping between slide narrative backbones (in a tree structure) to Bach *et al.*'s layout patterns [11]. G denotes a cell group, B denotes all the bulleted points in a group, M denotes a cell output such as a plot or table, and the gray circle denotes the root node of the tree.

completely customized slide manually, similar to traditional presentation tools, Slide4N supports this flexibility by offering a set of slide templates with five different layouts (*i.e.*, title slide, title with one-column content, title with two-column content, title only, and blank). The user can just click the "down arrow" on the *Control* (Figure 5.1C) to access them.

To support easy customization, Slide4N also offers interactive linking between the slides and their associated cells (G1, G5), which is shown in Figure 5.1 (blue bidirectional arrows). After selecting a slide in *Content* (Figure 5.1D) or *Slide* (Figure 5.1E), *Code* (Figure 5.1B) highlights the associated cells (pink dots) and *Control* (Figure 5.1C) is also updated to show the current configuration for the selected slide. This interactive linking not only allows users to easily trace the source codes to adjust potentially inaccurate code summaries, but also provides them with the flexibility to update the associated slide after adjusting the codes. In addition, an outline of all the slides is displayed on the *Content* (Figure 5.1D)

using the slide topics and titles. This can be used for navigating the slide deck, similar to most slide-making tools. There is also a navigation switch to control the display of navigation information on each slide, including a navigation bar and a page number. For example, on the right of Figure 5.6c, the navigation information is created by using the topics of the generated slides and the number of slides in each topic, and is placed at the bottom of the slides.

5.5 Slide4N User Interface

With the aforementioned back-end, we could locate relevant cells, distill titles and bullet points from code and markdown cells, and group and order generated contents. The user interface was developed through an iterative design process: we first derived design goals from the literature review, built low-fidelity prototypes, and then refined the interface over several rounds of internal feedback and pilot testing before the formal user study. Powered by the backend and guided by the design goals, we designed a novel user interface for Slide4N, which is implemented as a JupyterLab extension to provide a seamless experience for data scientists to create slides from computational notebooks.

As shown in Figure 6.2, the user interface(UI) consists of four interactively-linked components: (1) *Code*: link notebook and slides (**G1, G2**), (2) *Control*: configure the slide generation model (**G3, G5**), (3) *Slide*: render, refine and customize generated slides (**G4, G5**), and (4) *Content*: overview and jump between slides (**G1, G5**).

5.5.1 *Code*

Code concisely displays the cells of the notebook and their states, linking the notebook and Slide4N, allowing a user to select cells used for slide generation.

In *Code*, each cell of the notebook is visualized as a rectangle with equal width, the more content the cell contains the higher the rectangle is, and rectangles are placed from top to bottom to align with the order of the cells in the notebook. The dot on the left of the rectangle marks the state of a cell with color, including default, focused and selected, and the red border marks the cell being updated. The # below the dot indicates the markdown type of the cell, so that users can quickly distinguish different types of cells. The colors in this view are designed to be consistent to reduce the user's cognitive burden.

The following interactions are designed to help the user explore and select appropriate cells to generate the slides. **Hover**. When the user hovers over a cell, a tooltip pops up and

gives the user a quick overview of the cell content, including the cell type, the first 3 rows of codes, and the output image or table (if any). Also the cells associated are connected to each other by blue arcs. **Click**. When the user clicks on a cell, the associated cells will be highlighted in blue(if any), the darker means the greater the association, and the current cell is always the darkest. At the same time, the notebook will jump to the corresponding cell. **Double-click**. When the user double-clicks on a cell, the cell is selected to generate slide content, double-click again to unselect it. **Highlight**. The color of the dot will respond to the above three mouse events, hover and click correspond to the focus state, double-click corresponds to the selected state. Additionally, when the content of a cell is updated, the border of the corresponding dot will be highlighted in red.

5.5.2 *Control*

Control allows the user to specify the topic for selected tells, whether to automatically merge the associated cells as well as the expected level of details for the generated slides. The Generate Button is used to send these configurations to the backend to make effects on the NLP model, and take the generated content back to the UI. In addition, there are two buttons at the top of the control panel for creating AI and template slides(5 templates are currently supported), with the color of the button marking the slide type.

5.5.3 *Slide*

Slide renders the content returned from the backend in a proper layout. It is divided into title area(top) and content area(below), which uses a grid-based approach to place bullet points, images, tables, etc., all of which can be individually dragged and dropped to adjust the layout. The panel draws on the PPT design and provides a set of accessible interactions for users to refine and customize the generated slides. For example, users can add new bullet points, select related generated points and modify the points using Markdown-like grammar; insert, resize, and restore images; order, delete, and restore slides, etc.

5.5.4 *Content*

Content displays an outline of the slides using the slide topic and title, and the navigation switch is used to control the display of navigation information on slides, including navigation bar and page number.

5.5.5 Interactive Linking

For all the slides, Slide4N supports interactively linking them to their associated cells. As shown in Figure 5.1, by selecting a slide in *Content* or *Slide*, *Code* highlights the associated cells (pink dots). The original notebook can also be dynamically linked by clicking on the cell in *Code*. Moreover, by activating a cell in the notebook, *Code* highlights the current cell and the cells associated with it. This interactive linking not only allows users to easily trace the source codes to adjust potentially inaccurate code summaries, but also provides them with the flexibility to update the associated slide after adjusting the codes.

5.6 User Study

To evaluate our approach, we conducted a user study in which participants were asked to use Slide4N to create slides from a notebook. Our goals for this study are: (S1) assess the usability of Slide4N in supporting presentation slides creation from computational notebooks, (S2) understand users' behaviors when creating presentation slides from computational notebooks with Slide4N, and (S3) collect users' attitudes and feedback on Slide4N's support for creating presentation slides from computing notebooks. We did not conduct a comparative study since there are no appropriate, widely-available baselines. Currently, commercial presentation software such as Microsoft PowerPoint and Google Slides is commonly used by data scientists; however, these tools require much tedious and laborious work as mentioned earlier, which results in an unfair comparison in terms of our study tasks and goals. The closest system to ours is NB2Slides [230] which still has many differences as discussed in Section 5.2.1. It uses a template-based approach to generate slides fully automatically with little human intervention and has a set of very strict requirements on the input notebooks; whereas, in our study, we focus on investigating the human-AI collaborative aspects in slide creation without much constraints on inputs. Further, it is impossible to replicate their approach without the data and trained model. Thus, we decided to qualitatively assess users' experience of Slide4N in comparison to their current practices.

5.6.1 Study Design and Data

Our study contained two parts, where in Part 1, participants used a notebook provided by us to create slides with Slide4N, and in Part 2, they brought their own notebooks. In a more controlled setting, Part 1 enabled us to evaluate Slide4N with exhaustive types of

cells and topics in the data science workflow, because a user-created notebook may miss certain aspects. In a more realistic setting, Part 2 allowed us to assess the flexibility and generalizability of Slide4N. This way, we could investigate the capabilities of Slide4N from multiple perspectives more thoroughly. The entire study was conducted remotely via video conferencing software. Thus, we deployed Slide4N's back-end and front-end on Google Cloud to ensure the participants could access it via their computers.

Specifically for Part 1, our experimental notebook consisted of 32 code cells and included a complete data science workflow. The notebook combined two notebooks from the Kaggle competitions House Prices Prediction[2] with the goal of building models to predict the price of each house. We chose this competition because it is one of the most popular competitions on Kaggle and many data science courses use this competition as a tutorial. We combined two notebooks to form a data science workflow with five stages (*i.e.*, Introduction, Data, Model, Model Performance, and Conclusion). We also made some minor edits to make the notebook more concise and readable for participants, including deleting some redundant and duplicate cells. Note that these changes were not intended to improve the quality of the generated slide, but to guarantee the user study would not be too long.

5.6.2 Participants

For Part 1 of our study, we recruited 12 participants (10 men, 2 women; self-reported gender; aged $\text{M} = 23.5$, $\text{SD} = 3.47$) through social networks and word of mouth. The participants were all professional data scientists or software engineers with previous experience conducting and presenting data science projects, as well as familiar with Python and Jupyter Notebook. On a 1-7 Likert Scale, the medians of participants' self-reported familiarity with Jupyter Notebook, Python, and Machine Learning algorithms were 6, 5.5, and 5.5, respectively (1 = "no experience" and 7 = "expert"). The information collected from the participants also showed that on average, our participants make 5 data science-related presentations per week, as part of their work routine, indicating that our participants were familiar with creating presentations to present their data science projects.

In Part 2 of the study, through social networks, six new participants were recruited (3 men, 3 women; self-reported gender; aged $\text{M} = 24.1$, $\text{SD} = 2.9$). Among them, four had master's degrees and the rest had doctoral degrees in fields related to data science, such as computer science and statistics. There were five participants who identified themselves as experts in machine learning and one participant as a specialist in data visualization, indicating that all of the participants had sufficient professional data science and coding

[2]https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques

background. With the same Likert Scale as above, the medians of participants' self-reported familiarity with Jupyter Notebook, Python, and Machine Learning algorithms were 5, 6, and 5, respectively. Also, our participants had sufficient experience in generating data science-related presentations by creating 4.8 data science-related presentations per week, on average.

5.6.3 Procedure and Task

The procedure and task for Parts 1 and 2 of the study were the same. After participants used their own computers to join our online study session and gave consent, we shared the URL for accessing Slide4N. During the study, participants first completed a brief training session to get familiar with Slide4N by watching a short video and using a sample notebook to explore the system and generate some slides. Then, based on a given experimental notebook (Part 1) or one of their own notebooks (Part 2), participants were asked to create a slide deck to present what the notebook does by imagining their audience as technical and non-technical members of their teams. We instructed participants that the slide deck should consist of at least five slides based on their preferences and cover as many stages of the workflow in the notebook as possible. They were given about 30 minutes to create the slide deck but we did not put a hard cut-off. Participants were asked to use the AI option of Slide4N to generate the slides for sections "Data", "Model", and "Model Performance" if corresponding content was included in the notebook, and use the manual template option for sections "Introduction" and "Conclusion", which is usually not included in the notebook. In this way, they could have both automatic and manual slide creation experiences with Slide4N. They were free to refine the slides, such as editing the title and bullet points, and adjusting the layouts.

After, participants completed three post-study questionnaires about their impression of Slide4N, followed by a semi-structured interview. The three questionnaires requested them to rate their created slides, the back-end of Slide4N, and its user interface, respectively (see Figure 5.4 for questions). The questionnaire items were developed iteratively during pilot testing, drawing on standard usability dimensions [22]. We refined the items across two pilot rounds to ensure they captured the key aspects of slide creation with our tool. The interview included topics on their experiences with Slide4N, practices with existing tools, qualitative comparison between Slide4N and tools they used before, as well as general opinions towards human-AI collaboration. Detailed questions in our study can be viewed in the supplementary materials. In total, the study session lasted about an hour and participants were remunerated by $15.

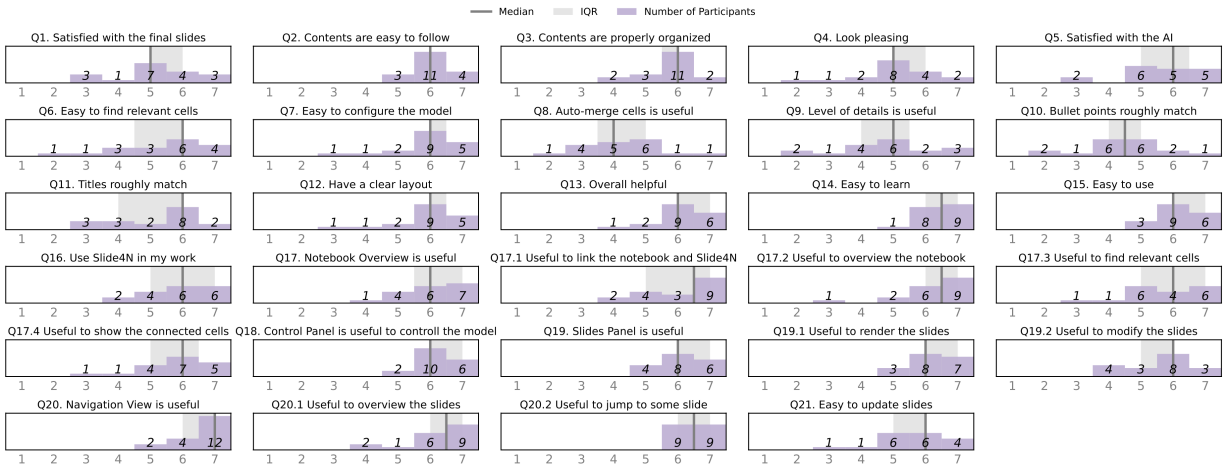


Figure 5.4: Participants' ratings on the post-study questionnaires including the created slides (Q1-4), the back-end (Q5-12), and the user interface (Q13-21), on a 7-point Likert Scale (1 = "strongly disagree" and 7 = "strongly agree").

5.7 Results

Overall, 17 out of 18 participants successfully completed the task within 30 minutes, and one participant took 35 minutes due to intentionally trying out all of the system's features. In the following, we report our user study results including participants' questionnaire responses, quality evaluation of the generated slides, and qualitative feedback to the system.

5.7.1 Questionnaire Responses

We designed three post-study questionnaires to evaluate the created final slides, Slide4N's back-end, and its user interface, revealing insights into our study goals S1-3. Figure 5.4 shows the distributions of ratings as well as the means and IQRs for all the participants in study Parts 1 and 2. The results from the two parts are very similar; please see our supplementary materials for additional figures.

Q1-4 regard participants' impressions with the final slides generated by Slide4N. Overall, the participants were satisfied with the final slides (Q1), with a median rating of 5 ($IQR = 1$). They all thought the contents (titles and bullet points) on slides were easy to follow (Q2; $M = 6$, $IQR = 0$) and properly arranged (Q3; $M = 6$, $IQR = 0.5$). The rating for the aesthetics of the final slides (Q4) was relatively low ($M = 5$, $IQR = 1$), but still positive. Two participants (P3 and P5) gave negative ratings. With our follow-up interviews, we

found out this was due to personal preference in creating slides, with some people preferring a concise style and others preferring to add more stuff (*e.g.*, bold, underline, and colors).

Q5-12 show participants' impression of the usefulness of the automatic slide generation (*i.e.*, the AI) of Slide4N. Overall, participants were satisfied with the experience of making slides using the AI-Assisted tool (Q5; $M=6$, $IQR=1.5$), except that P5 and P18 rated 3. By observing P5's other ratings, we found this was actually because of the layout adjustment and bullet points editing, and he also suggested Slide4N to support some keyboard shortcuts (*e.g.*, copy and paste, redo and undo) and more fine-grained layout adjustments. P18 wanted to have more bullet points for cells with many lines of code. More specifically, the participants thought they could find relevant cells (Q6; $M=6$, $IQR=1.5$), the model is easy to configure (Q7; $M=6$, $IQR=0.5$), the generated bullet points (Q10; $M=4.5$, $IQR=1$) and titles (Q11; $M=6$, $IQR=2$) roughly matched what they wanted to put on the slides, and the generated slides had clear layouts (Q12; $M=6$, $IQR=0.5$). However, P3 and P11 found it hard to find relevant cells (Q6), citing that they didn't fully understand the visual encodings in *Code* (Figure 5.1B). In addition, Slide4N's features, "Auto-merge" (Q8) and "Level of details" (Q9), were rated relatively lower; however, the medians remained positive. By analyzing the recorded videos, we found most participants used the default settings, indicating they may not be aware of the impact of these two configurations.

Q13-21 reflect participants' impression of the usefulness of the Slide4N user interface. Overall, Slide4N was perceived as very helpful (Q13), easy to learn (Q14), easy to use (Q15), all having a median rating of 6 or more. Although several participants rated low on some aspects of Slide4N, they all wanted to use Slide4N in their future work (Q16) and rated it fairly high (six rated it as perfect). P5, while giving more negative ratings on other questions, still rated 6 for Q16; he further commented, "Slide4N effectively links data analysis work with the presentation slides, it can save me a lot of time in creating slides, although Slide4N should be improved to support an easier way to customize slides." Participants also felt that the four panels in Slide4N (Figure 5.1) were very helpful (Q17-20), all with a median of 6 or more. But P5 thought that *Code* (Figure 5.1B) was cluttered when there were many arcs, and thus rated 3 for Q17.2. P16, who rated 3 for Q17.3, thought that *Code* was not easy to select cells because the notebook he used was very long and the rectangles were sometimes too small (Figure 5.6a). Almost everyone thought that *Control* (Figure 5.1C) was useful for controlling the slide generation model (Q18). *Slide* (Figure 5.1E) was rated relatively low, probably because the user interactions provided by Slide4N for editing the bullet points and adjusting the layout could not be on par with professional software such as Microsoft PowerPoint (Q19.2). All participants thought that *Content* (Figure 5.1D) was useful for slide overview (Q20.1) and navigation (Q20.2).

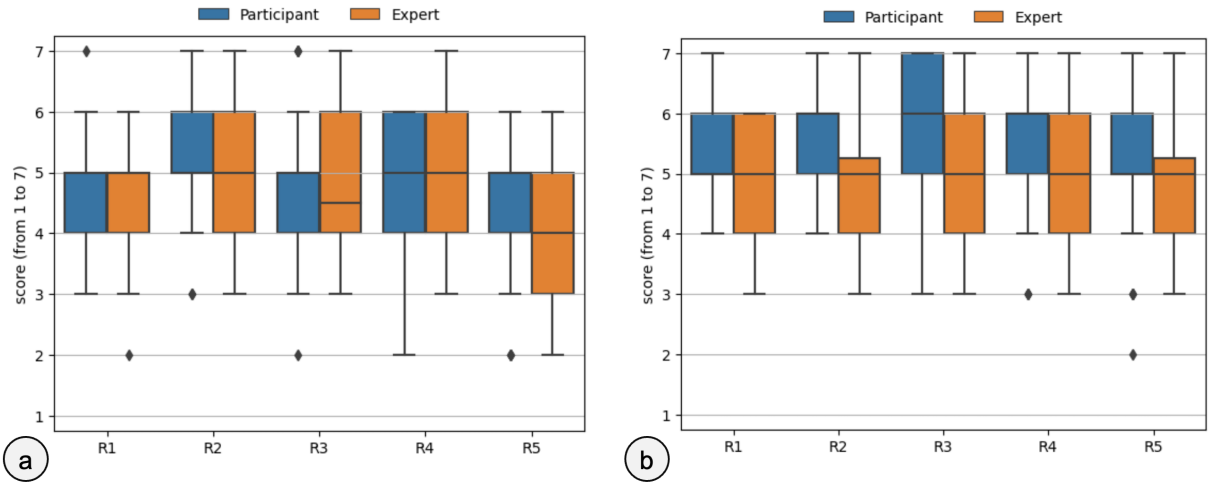


Figure 5.5: Audience ratings on five aspects (R1-5) of the slides created by Slide4N: (a) boxplots of ratings for the 12 decks of slides created using the experimental notebook in study Part 1, and (b) boxplots of ratings for the 6 decks of slides created using participants' own notebooks. The five aspects include overall satisfaction (R1), clarity of structure (R2), ease of understanding the content (R3), proper organization of the content (R4), and aesthetics (R5), on a 7-point Likert Scale (1 = "strongly disagree" and 7 = "strongly agree").

Updating slides was also perceived easily (Q21), except that P3 rated 3. Combined with the interview, we found that this was caused by Slide4N hiding the image he pasted onto the slide after the update.

5.7.2 Slides Quality Evaluation

While Q1-4 in our questionnaire indicate participants' self-impression of the created slides, these do not reflect how the slides are received by the audience. As part of our evaluation plan described below, to assess the quality of the slides created with Slide4N more comprehensively, we invited the six participants in Part 2 to rate all the slides in the whole study (except their own slides); thus each of them rated 17 decks of slides. We did not invite the participants in Part 1 for the slides rating because they all created slides from the same experimental notebook, which might generate bias. Additionally, we invited two experts, who are university professors with years of experience in preparing, delivering, and judging presentations, to rate all the 18 decks of slides created in Part 1 and Part 2.

The results of their ratings are shown in Figure 5.5, on five aspects including overall

satisfaction (R1), clarity of structure (R2), ease of understanding the content (R3), proper organization of the content (R4), and aesthetics (R5). These five aspects were chosen based on prior work on presentation quality assessment [27] and refined during pilot testing to capture the dimensions most relevant to notebook-derived slides. For the slides created in Part 1 (Figure 5.5a), most of the median ratings were 5 or above for both the participants and experts. The experts rated the ease of understanding slightly lower ($M = 4.5$). Another exception was the aesthetics which received the lowest rating from the experts ($M = 4$); however, in this chapter, we did not focus on the aesthetic capabilities for Slide4N. This could be easily enhanced by employing themes and templates like those in commercial presentation tools. For the slides created in Part 2 (Figure 5.5b), the overall median ratings were higher, where all aspects were 5 or above. This might be due to the fact that participants were more familiar with their notebooks and the contents were richer, which influenced the quality of the created slides. In summary, the ratings from the six participants and two experts indicated the overall effectiveness of Slide4N and its stable performance in aiding slide creation from notebooks.

5.7.3 Qualitative Feedback

Our interview mainly focused on collecting participants' feedback on all aspects of Slide4N, practices with current tools, and human-AI collaboration, to assess S1-3 qualitatively. In general, participants appreciated our human-AI collaborative approach to creating presentation slides compared to the fully-manual or fully-automatic methods. In the following, we report our results on the challenges of current presentation software and their feedback on Slide4N.

5.7.3.1 Challenges of Current Presentation Software

All participants said that Microsoft PowerPoint was the tool they mostly used to report on their data analysis work. The process of creating slides with PowerPoint can be roughly divided into three stages: (a) write an outline to sort out the content to be reported or find a template that has been used before, (b) distill key content from the code and transfer it to the slides, and (c) style the slides to make them more presentable. They agreed that using PowerPoint to report data analysis work had the following problems. First, it was difficult to find relevant cells from a long and messy notebook, as P1 said: "As the notebooks got longer, I couldn't remember the interconnections between the cells and had trouble finding them." Second, distilling presentable content from the notebook was time-consuming. "After the analysis, I usually spend a lot of time thinking about the

content of the presentation." (P9). Third, organizing, laying out and styling the content on the slides was also tedious. Fourth, when the code was modified, the corresponding slides needed to be located and updated manually, a process that was error-prone and laborious, as P14 said: "it's harder to change slides than to make them." Besides, most participants were willing to clean up the notebook before making the slides, such as deleting excess codes and adjusting the cell order, but almost no one is willing to add documentation. "Documentation helps me understand the notebook, but it's tedious and time-consuming to do so." (P1).

The result is consistent with our expectations on the deficiency of current practice in presenting data science work conducted with computational notebooks. It also validates our design goals and the development of Slide4N. Further, it extends our vision of closing the analysis presentation divide.

5.7.3.2 Feedback on Slide4N

The feedback from all participants on Slide4N was highly positive. In general, participants thought Slide4N's user interface is clear and good-looking, and they were more willing to use our tool than commercial tools such as Microsoft PowerPoint, especially when their time is limited. In the following, we group their feedback based on our design goals in Section 5.3.1.

G1: Link the notebook and presentation slides. When asked about the advantages of Slide4N, almost all participants mentioned it was very convenient to link notebook cells and slides. Most of them felt this helped structure and organize their presentation slides. On one hand, this allowed them to quickly adjust and update the corresponding slides after changing the code. "I can quickly go back to the source code I referenced when I created the slides [for adjustments]." (P1). "Slide4N relieves me from the burden of remembering the association between slides and notebook cells, which is needed for updating." (P4). "This saves me the time to copy and paste the cell output image." (P12). Also, participants suggested some improvements to Slide4N. "I would like to link bullet points to cells, which would make customization easier." (P8). "Currently it supports synchronization of slide content after code changes, but I would like to keep my layout while updating." (P6). On the other hand, they believed it made communication within the team easier, as P1 and P16 mentioned "Slides are easier to accept and understand, while Slide4N allows me to quickly link to my codes when I discuss technical details with developers." In addition, participants thought the link helped them organize their notebooks, as P17 said: "I don't have to organize my notebooks linearly to facilitate the creation of the slides, while I believe *Code* can also assist me with this." Furthermore, three participants mentioned that

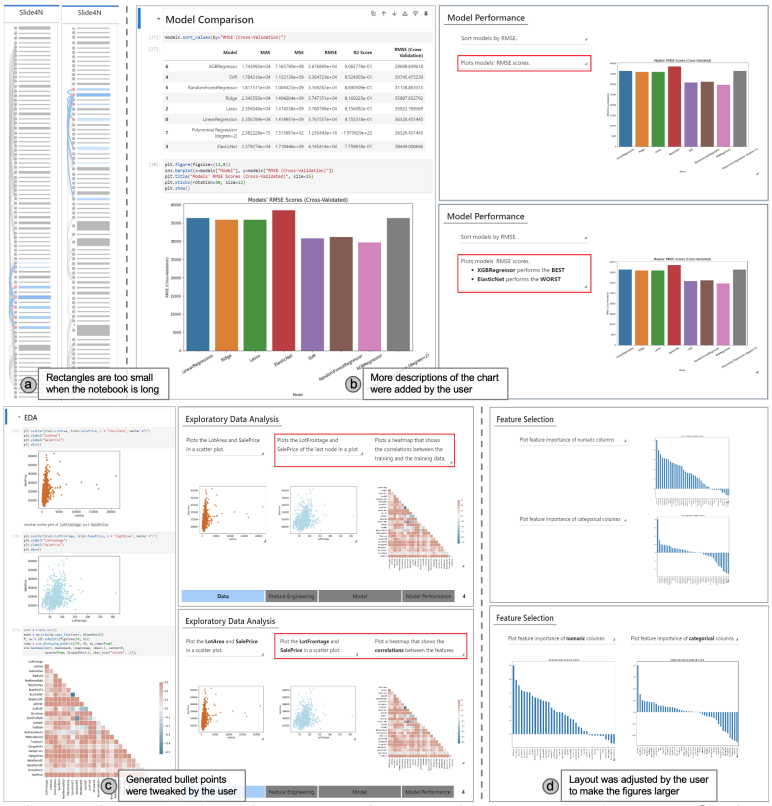


Figure 5.6: (a) When the notebook is too long, the rectangles in *Code* can be small and difficult to select. (b)(c) Slide4N sometimes generates less satisfying bullet points and the participants needed to improve the text (indicated by the red boxes) on their slides (left: the original notebook; top-right: AI-generated bullet points; bottom-right: edited bullet points). (d) Slide4N sometimes produces undesired layouts and the participant wanted to make the figures larger.

building Slide4N into JupyterLab made the link even tighter. "I can do data analysis and slides in one window." (P16). "The process [to create slides] is simpler and smoother, and it saves me the time switching between PowerPoint and the notebook." (P18).

G2: Help locate relevant cells in the notebook for creating slides. For locating relevant cells, P1 appreciated *Code* for doing so, "I do like *Code*, and I can quickly find relevant cells in my messy notebook." However, P13 and P15 suggested that when computing cell relevance, it is insufficient to only consider variables with the same name, but also the data they are associated with. For the design of *Code*, P17 appreciated it very much: "I can intuitively see the selected cells, their positions, and some details of the cells,

which helps in the overall understanding of the notebook." Meanwhile, participants offered some suggestions. When selecting cells, P8 suggested supporting a one-click selection of relevant cells, which was envisioned at the beginning of our design. However, it might lead to unclear associated cells for slides, less predictable generated content, and users might spend much time adjusting the generated content. P2 and P8 suggested reflecting the hierarchy of the notebook more obviously (*e.g.*, using dotted lines to separate different parts), "It will allow me to quickly understand the structure of the notebook, which facilitates the organization of the report." (P8). P7 and P16 (who have light color vision deficiency) recommended using larger dots or brighter colors to mark cell states, "because it would be more obvious when there are too many cells." (P7).

G3: Assist with distilling key information from code and Markdown cells. All participants thought having AI to assist them to distill titles and bullet points from the code is very convenient. As shown in Figure 5.6c, P9 and P17 mentioned "AI gives me a good 'first-cut.' I just need to tweak the generated content a bit to my preference and it can be used for the report." We also found that participants had different requirements for the generated bullet points. Most of them felt that the distilled points roughly matched what they want to put on slides. Two expected the distilled points to answer not only what was done, but also how it was done. Regarding the topic, most of them thought the options we provided were sufficient for reporting. While P1 suggested that the "Data" can be subdivided, "because I spend most of the time doing data processing, and there is a lot to tell about data." The interfaces of Slide4N allow users to provide high-level guidance for slides generation: "Auto-merge cells" and "Level of details". We found that most participants explored these options first but later adopted the default settings, which generated more detailed content for the slides. But P15 preferred the settings to generate more concise content. In addition, four participants thought that some options for configuring the model could be added to guide the model. P7, P13, and P17 jointly suggested allowing users to set their preference for generating content from code or documentation. P18 thought that more bullet points should be generated for cells with many lines of code to provide more details, so he suggested providing a configuration that automatically breaks down long cells. Regarding the time delay of the generation, almost all participants could accept it.

G4: Arrange slide contents in a logical and coherent narrative flow manner. All participants felt that the organization of the content on the slides matched the logical correlation between them. All praised the initial layout provided by Slide4N. "I like the generated layouts, I don't need to spend much time thinking about how to organize the content on the slides and keep the style consistent between slides." (P12). "The generated layouts save me a lot of time, and a simple tweak to them can satisfy my layout needs." (P11). We also found that participants' layout preferences varied. Most preferred

a consistent layout style, while P14 wanted more diversity of the generated slide layouts, "because it draws the audiences' attention to the presentation." In addition, P14, P19, and P20 specifically mentioned that the alignment of elements on the slides was important. Three participants also made suggestions for the layout. P5 and P13 preferred a more fine-grained grid "because I can be more flexible in adjusting the layout." P7 wanted bigger images (Figure 5.6d) because "when I place images on slides, I tend to care more about the images than the texts." He also suggested enabling users to specify the layout of the generated slides. In addition, three participants appreciated the navigation provided by Slide4N. "Navigation allows audiences to quickly know the current progress of the report, while the page number is convenient for Q&A." (P10). "This is what I want to put on the slides, but PowerPoint does not support one-click generation, and adding it manually is laborious and time-consuming." (P7), which was also agreed by P20.

G5: Support necessary human intervention for slide generation. Slide4N provides necessary human intervention for users to customize the generated slides, which can be roughly divided into three stages: select the inputs (cells) to the slides generation model, configure the parameters of the model, and adjust the content generated by the model. All participants thought such human-AI collaboration was acceptable and met their needs for AI. "Slide4N meets all my requirements [for AI]. It is very comfortable to use." (P9). "The human intervention provided by Slide4N is necessary and it helps me with the tedious and basic things, so I can spare some time to focus on how to tell the story effectively." (P10). Besides, participants preferred a human-AI collaborative approach compared to a fully-manual or a fully-automatic method to create slides. As we said in Section 5.7.3.1, all participants thought that the fully-manual approach was tedious and time-consuming. For the fully-automatic approach, all held a negative attitude. As P9 said: "I don't trust the black-box model, and I want to intervene in the generation process." P16 and P18, who agreed with P9, said: "The generated contents can't be predictable, and it's hard to adjust when the slides are not what I want, like the structure and [slide] contents." P1 also said: "I don't believe today's AI can automatically generate high-quality slides. Realistically, I don't want to have such an AI either, because I don't want to lose my job." Moreover, we believe today's AI can't go beyond the notebook, so some slides (*e.g.*, background and limitations) can't be provided by AI, and human involvement is needed. When asked what they thought about creating slides one by one, all gave positive feedback, as P6 said: "I can have more control of the slide creation process, the content generated by the model is more predictable." P11 said: "It's easy for me to incorporate my own ideas into the slides, which is very important for a presentation."

5.8 Discussion

5.8.1 Design Implications

From our user study, we consolidate a few design implications that could shed light on the future design of similar tools.

5.8.1.1 More Adaptive Slides Generation

Slide4N allows users to manage the level of detail and complexity of the slides. However, depending on the scenario in which the slides will be used, data scientists may have different requirements for their slides. The designers of these tools should therefore take a user-specific approach in terms of providing different types of content and different layouts to generate the slides. For example, participants suggested that it is an advantage for Slide4N to generate complex slides that contain in-depth descriptions with more complex visualizations from the notebooks for those who need to present in-depth reports or research proposals, as well as creating slides with simple figures and less complex contents for those unfamiliar with the notebook, especially non-technical individuals or students. We recommend that future AI-assisted slide creation tools should generate presentations adapted for various audiences by collecting slides in different usage scenarios and fine-tuning the content generation model. Participants also suggested the generated bullet points can cover how the code got the result instead of just describing what the code does. For example, inline code comments can be used to recommend more alternatives for the generation.

5.8.1.2 More Obvious Data Provenance

Creating slides often requires some materials (*e.g.*, text, images, code, etc.) which are *data provenance* for slides; in this chapter, it mainly refers to the notebook cells. Participants found it very helpful to bind the slides to the cells when asked about data provenance. Such bindings can quickly locate the original cells, which 1) simplifies the modification and updating of slides, and 2) provides more detailed information for presentations to accommodate different scenarios, such as using original code to facilitate technical communication, and using interactive visualizations for more vivid upward reports. Additionally, bindings can assist the user in locating incorrect annotations in the notebook, which can be adjusted to make it easier to comprehend. In this regard, we can argue that slide creation tools could integrate the binding feature between slides and the original materials used for

slides (*e.g.*, code, text, and images) to enable the users with a more flexible and interactive slide creation experience.

5.8.1.3 Better Human-AI Collaboration

When creating slides with Slide4N, the AI can assist data analysts in three aspects: 1) selecting cells as input to the model, 2) configuring the model at a high level, and 3) adjusting and customizing the generated content. By repeating the above process, users can quickly create the slides they want one by one. Such Human-AI collaboration not only simplifies the process of creating slides and enables users to participate more actively but also ensures that AI-generated content matches users' expectations and is customized according to their needs. Combined with the results of the user study, we believe that there is room for improvement in all three aspects.

For input, Slide4N currently supports selection at the cell level, future extensions can be made to support selecting at the sub-cell level, which can leverage the location of inline comments and the functional scope of code. With *Code* (Figure 5.1B), we can apply brush interaction to facilitate this feature. It would be beneficial to allow the user to control the threshold of relevance, such as the relevance score (see Section 5.4.1) and the number of shared variables, which not only allows users to filter for highly relevant cells but also reduces the clutter caused by many arcs when there are many relevant cells.

Regarding model configuration, other effective high-level controls can be provided to the user, suggested by our participants from the user study. For instance, users may adjust the model's preference to accommodate different presentation scenarios by taking into account analysts' programming habits (*e.g.*, rich or sparse annotations) and the intent of the generated content (*e.g.*, inform, convince, tell a story). However, when users have multiple configurations, they may be overwhelmed and do not know what to do. To avoid unnecessary time overhead, it is essential to clarify to the users the possible impact of the configuration (*e.g.*, provide metrics to measure the impact or in-situ instructions) or weighted merging of multiple related configurations into an integrated one.

As for customization, the AI should learn user behaviors for adjusting slides, including wording style, bolded text (often for content that needs to be emphasized), preferred layout methods, etc., and provide suggestions on how to refine future slides similarly. Furthermore, the AI should be able to extract templates from user-created slides and add them to the user's personal template library, allowing for customized layout manipulation. Additionally, given that visualizations are common parts of such presentations, and that analysts may find it bothersome to do so (*e.g.*, which toolkit to use, what APIs are avail-

able, and how to use them), we believe it would be useful to recommend and add relevant visualizations to the slides, based on the code. The recommendation can be supported by several existing tools built in the Jupyter Notebook (the classic notebook interface compared to JupyterLab) [45, 211].

5.8.1.4 Less Bias and Risks

The story we tell, in this case, data science work, is influenced by our personal background and life experience. Therefore, we may not realize that we are limiting ourselves to certain aspects and ignoring others that may be meaningful (we call this personal *bias*), and the impact is often detrimental. The results of our user study show that Slide4N can reduce users' personal bias when making slides. Our analysis determined that two participants (P16 and P17) were inclined to choose cells containing plots to make slides, thus ignoring other cells (*e.g.*, long code cells). This implies that users may be reluctant to digest long code cells, while our AI does not differentiate between cells of different lengths in digesting code. To resolve this issue, *Code* may be enhanced to highlight used cells, which will facilitate the user's discovery of unused cells. In addition, people tend to tell stories from certain fixed perspectives, which may lack appeal to audiences, especially in similar scenarios, but AI can provide some new perspectives that will guide users to create slides with more comprehensive and diverse content.

Although the AI component of our system is designed to assist data analysts in creating slides, there are some potential *risks* with this process. First, our analysis of participants' feedback indicates that privacy is one of the main concerns of our participants. Three participants suggested that Slide4N should run locally when sensitive data is involved so as to prevent the leakage of data. While our system relies on online tools for implementation, further improvements may be considered to enhance user privacy, by encrypting their data as well as executing Slide4N on the user's local computer. Second, users may over-rely on the automation solution provided by Slide4N, leading to their inexperience with slides and thus affecting the quality of presentations. Considering our system is designed to create slides with human-AI collaboration and requires user interaction, we believe that the negative impact is relatively insignificant. In addition, the user study revealed that participants integrated their own elements (*e.g.*, insights and styles) into AI-generated content, which led to a greater familiarity with the slides. But future study needs to further explore the potential over-reliance on AI.

Another concern is that slides produced with AI assistance may lack variety across different users, because the underlying model tends to generate similar structures and phrasings. To mitigate this, Slide4N provides multiple configuration options and allows

users to edit all generated content. Future work could introduce template diversity and style variation to further reduce this risk.

A potential concern is that the evaluation in this chapter focuses on usability and user satisfaction, which may appear to lack a theoretical contribution. We view the contribution as threefold. First, Slide4N demonstrates a design pattern, cell-level content extraction followed by template-based assembly with user refinement, that can generalize beyond notebooks to other structured documents. Second, the study provides empirical evidence that the Propose–Inspect–Refine collaboration loop, introduced in the thesis introduction, works in the presentation domain: users preferred the AI-generated starting point over a blank canvas and made targeted edits rather than wholesale rewrites. Third, the findings reveal specific boundary conditions: AI-generated slides worked well for content organization but required human judgment for narrative coherence and audience adaptation. These results extend our understanding of when and how human–AI collaboration benefits artifact creation tasks.

5.8.2 Limitations and Future Work

There still exist several limitations in our tool and user study. We outline them below and provide directions for future work.

5.8.2.1 Slides Generation Model

Slide4N only generates bullet points based on each code cell, not including Markdown cells or code outputs. Markdown content is often considered as a summary of a section, which can provide a supplementary explanation for code. Code outputs can be used to explain the code purpose and method in detail. The system can be improved by conducting an empirical study to investigate the relationship among code cells, code outputs, and markdown cells and figure out how to make good use of such information. Moreover, code outputs in Jupyter Notebooks can be complicated, often including charts and tables, which are leveraged by our method to generate visually appealing slides. However, vision-impaired people would need to rely on alternative text or the help of a screen reader [143]. To make the generated slides more accessible, inspired by the work on chart and table summarization [57, 70, 77, 210], future work may explore automatic methods that generate alternative text for the charts and tables on the slides, which will broaden the audience and use-cases for Slide4N. As shown in Figure 5.6b, this is an example where the participant enriched the bullet points based on the chart. Also, even though most participants were satisfied with

the slide generation, several participants were still not very satisfied with the slide content. Future work can be done to build a specific dataset for the slide creation task and train the model in a large repository.

5.8.2.2 User Interface

The front-end of Slide4N can be enhanced. When creating slides, we usually need to consider the structure (often refers to the outline), content, layout, and styling. Slide4N can help users distill the key bullet points and titles from notebook cells, render these contents with appropriate layouts and support further adjustments. Thus, we believe Slide4N can effectively address the needs for content and layout. However, Slide4N provides limited support for structure (*e.g.*, selection and creation of topic, which is similar to the paper section). Five participants indicated that they would reuse the structure of previous slides in similar presentations, which is also pointed out by [153]. As a result, Slide4N should enable users to extract structures from existing slides to reuse them again. As for styling, Slide4N currently only supports markdown-based editing, which we believe is often insufficient for formal situations (as five participants stated in the user study, while they liked the preset styles provided by Slide4N). Future improvements may include support for more styling features, such as colors, bolding, font size, etc. In addition, providing in-situ guidance for the users and supporting export to other formats (*e.g.*, .pptx) will improve Slide4N and address some user needs.

5.8.2.3 Evaluation

Our user study also has limitations. First, in this chapter, we mainly focused on designing and developing the system, and we did not choose a baseline to compare in our study as explained in Section 5.6. However, a thorough empirical investigation comparing different families of approaches would help gain a broader understanding of the advantages and disadvantages of these different approaches. These approaches include deep coupled human-AI collaboration (*e.g.*, Slide4N), fully-automatic generation (*e.g.*, NB2Slide [230]), and fully-manual operation (*e.g.*, Microsoft PowerPoint). Second, we evaluated Slide4N using both a provided experimental notebook and participants' own notebooks in study Part 1 and Part 2, respectively. This indeed provides insights into the usage of Slide4N in both controlled and natural settings. However, future work is needed to carry out a long-term deployment study with more participants to better assess the usefulness of the system in the wild without any constraints. Third, Our current study design is in a controlled environment with limited notebooks and task procedures, but slides creation tasks in the wild

can be diverse and flexible. Most of the participants in our user study were data engineers, but non-technical users such as managers also need similar support in creating slides for their work. Future studies need to be conducted with different groups of participants to examine the effectiveness of our system.

5.9 Conclusion

This chapter presents Slide4N, an intelligent and interactive tool built within JupyterLab to support the *presentation* stage by enabling data scientists to create slides from computational notebooks. By leveraging advanced natural language processing techniques, Slide4N distills titles and bullet points from user-selected notebook cells and arranges them algorithmically into groups, and displays them with appropriate layouts. A tool such as this automates many of the tedious tasks associated with the creation of slides, leverages appropriate user inputs, and produces results in a format that is widely used for presentations. A user study was conducted to evaluate Slide4N. The results indicated that it was both useful and effective in supporting slide creation tasks from notebooks as well as encouraging human-AI collaboration. Our study also contains some design implications that will inform future tool development.

Chapter 6

Human–AI Collaboration for Multi-Dimensional Code Evaluation and Optimization

This chapter addresses **RQ(d)** from the introduction, focusing on the *evaluation and optimization* stage of programming workflows. The preceding chapters support understanding, documentation, and presentation; once programmers have code in place, the remaining step is to evaluate and optimize it. How does AI-driven feedback support programmers in evaluating and optimizing code quality within existing development workflows? We present MACEDON, an IDE extension that supports human–AI collaboration through real-time, multi-dimensional evaluation within the IDE. The system surfaces feedback across clarity, documentation, redundancy, runtime, and memory, invites iterative human decisions about which improvements matter, and guides programmers toward better implementations. Code quality involves trade-offs that cannot be resolved by a single metric; human–AI collaboration lets programmers inspect evidence and choose what to optimize.

6.1 Motivation

This chapter addresses the *evaluation and optimization* stage of the programming workflow introduced earlier (Figure 1.1). We develop an interactive human–AI collaboration tool that enhances this stage by surfacing multi-dimensional code quality feedback and supporting iterative optimization decisions.

Evaluation and optimization are a critical late stage of programming workflows: after code is drafted and iterated on, programmers still need to evaluate quality and make trade-offs across dimensions such as correctness, clarity, redundancy, documentation, runtime, and memory. This stage has become more prominent as modern code assistants make it easy to generate and revise code from natural language prompts [30, 52, 114, 220, 226]. Novice programmers, in particular, increasingly rely on such assistants to generate or optimize code in their workflows [78, 132, 185]. Yet producing a candidate solution is not the same as knowing whether it is a good solution: evaluating code quality requires contextual reasoning about intent, constraints, and what to optimize for.

Automation alone does not work well at this stage. One-shot assistant feedback can be hard to verify and may encourage surface-level changes that do not match a programmer's goals, while traditional tooling often surfaces only one dimension at a time and requires separate workflows to interpret results. As a result, programmers frequently rely on prompt iteration and ad hoc manual checks, which is inefficient and leaves them uncertain about whether improvements in one area degrade another [14, 115, 172, 185].

For this reason, we frame evaluation and optimization as a Human–AI collaboration problem. Rather than replacing programmer judgment, the goal is to make multi-dimensional evidence visible, connect it to concrete code segments, and support iterative decision making: users should be able to inspect the current status of their code, choose what to optimize, and refine changes in a controlled loop. Therefore, our paper explores designs that support the iterative process of assistant-supported code evaluation and optimization, where programmers and tools jointly iterate on evaluation and optimization decisions.

To address this gap, we first conducted a formative study involving six programmers with varying experience levels who regularly use LLM-based tools for code optimization. Our goal was to investigate how programmers assess their code and formulate optimization strategies in their existing workflows and the challenges they encounter. The study revealed a consistent need for *structured, multi-dimensional* code evaluation that surfaces aspects such as clarity, redundancy, documentation, time efficiency, and space efficiency. Participants noted that they often overlooked space usage or documentation quality unless specifically prompted. They also expressed the need to selectively optimize individual code segments without losing control over the broader codebase.

Building on insights from the formative study, we focus on exploring the practice and design of LLM-assisted code optimization and evaluation around the following research questions:

RQ1 - Evaluation & Strategy. *What evaluation dimensions and optimization strate-*

gies do programmers adopt when working with LLM-based assistants?

RQ2 - Tool Design & Effectiveness. *How can we design effective tools that support multi-dimensional code optimization, and to what extent are they useful for real-world programming tasks?*

RQ3 - Generalizability. *Can such tools extend beyond novice programmers and remain effective in more diverse programming scenarios?*

To answer **RQ1**, **RQ2**, and **RQ3**, we began by analyzing a large dataset of over 70,000 real-world code examples from the Performance-Improving Edits (PIE) dataset to identify five measurable dimensions for code optimization. This analysis provided a foundation for understanding how programmers evaluate and improve their code, answering **RQ1**. To explore effective designs for code optimization (**RQ2**), we developed MACEDON, a Visual Studio Code extension that supports multi-dimensional code evaluation and targeted, segment-specific suggestions. The tool was built based on three derived design goals: 1) helping users assess their code across the five dimensions using interpretable scores, 2) offering specific suggestions such as improving loop efficiency or renaming unclear variables, and 3) enabling users to apply changes consistently with minimal manual effort. To further examine whether this design is effective for real programming tasks (**RQ2**), we conducted a user study with 24 novice programmers. The results show that MACEDON helped participants evaluate their code easily and make more effective improvements. Specifically, compared to existing LLM code assistants, participants completed tasks more efficiently and produced code with higher expert ratings in clarity, maintainability, and performance. Furthermore, to explore whether MACEDON is useful beyond novice programmers (**RQ3**), we conducted two case studies in which participants applied the tool to real programming tasks. The case studies show that MACEDON supported users' goals, helped them organize their codebase, and remained effective in broader programming contexts.

In summary, our contribution is threefold:

- A formative study and a comprehensive data analysis of optimization behaviors and needs when using LLMs for code evaluation and optimization.
- An interactive human–AI collaboration tool, MACEDON, developed as a Visual Studio Code Extension, that supports segment-specific code optimizations in multiple dimensions, minimizing rework and improving efficiency.
- A user study and two case studies for assessing MACEDON, demonstrating its effectiveness in improving code quality and user experience compared to conventional LLM-based assistants.

6.2 Related Work

Section 2.3.4 reviews traditional and LLM-based tools for code evaluation and optimization. We distinguish our approach below.

6.2.1 Generating and Optimizing Code with LLMs

Recent programming workflows have shifted towards leveraging LLMs for both generating and optimizing code. LLMs like GPT [1], Codex [30], CodeGen [141], and InCoder [51] can interpret natural language instructions and produce corresponding code snippets [48, 154, 162]. These models allow developers to specify high-level generation and optimization goals through natural language prompts, reducing the time spent on manual adjustments. For example, tools like GitHub Copilot utilize LLMs to assist developers in code completion and optimization by understanding the surrounding code context and generating suggestions [14, 140].

Other LLM-based frameworks have also been developed to address different challenges in code generation and optimization. For instance, ClarifyGPT [133] enhances the code generation process by identifying ambiguities in user prompts and seeking clarifications, ensuring that generated code aligns closely with user intentions. Similarly, the use of reinforcement learning in conjunction with LLMs has enabled models to adaptively improve code based on feedback from test cases, further refining the generated outputs [43]. CodePlan [12] addresses the challenge of complex repository-level tasks by using a task-agnostic, neuro-symbolic framework that frames coding as a planning problem, synthesizing a multi-step chain of edits through dependency analysis, change impact analysis, and adaptive planning with neural LLMs. Additionally, methodologies combining program analysis with LLMs, such as leveraging GPT-4 for identifying inconsistencies between code comments and their implementation, have shown success in automating code documentation maintenance and reducing developer effort [225]. SBLLM [55] combines LLMs with search techniques for iterative code optimization, using representative sample selection, adaptive pattern retrieval, and genetic operator-inspired prompting to achieve code efficiency improvements. CoLadder [217] provides a hierarchical structure for decomposing programming tasks, allowing programmers to better align their problem-solving intentions with LLM-generated code, thus improving the optimization and modification of code across various abstraction levels.

While prior research has explored the use of LLMs for coding tasks, as well as frameworks that address ambiguity in prompts and complex repository-level tasks, our work

focuses on the iterative nature of the code evaluation and optimization process in natural language programming. Unlike search-based optimization [55] or task decomposition [217], our approach emphasizes providing programmers with real-time insights into their code's status through a structured interface that facilitates direct, segment-specific optimizations. MACEDON uniquely combines visualization of code state with targeted recommendations, enabling programmers to efficiently track and improve code quality through a seamless integration into their workflow. Additionally, our work fills a crucial gap in supporting programmers' understanding of their code before crafting optimization prompts, thereby offering a more comprehensive solution for iterative LLM-driven code optimization.

6.2.2 Evaluation of LLM-based Code Assistants

Evaluating the correctness and quality of code generated by LLMs is crucial for understanding their effectiveness in software development tasks. Traditionally, benchmarks like HUMANEVAL [30] have been used to measure functional correctness by running predefined test cases against generated Python functions. Similarly, MBPP [9] provides a set of crowd-sourced programming problems with corresponding test cases, allowing for a broader evaluation of LLMs' code generation capabilities. Expanding this to multiple languages, MultiPL-E [25] and HumanEval-X [231] translate these benchmarks to multiple other coding languages, enabling cross-language evaluations for models like Codex and CodeGen. For competitive programming, AlphaCode [114] uses complex challenges from Codeforces to test problem-solving capabilities, while Spider [218] assesses text-to-SQL conversion tasks. These benchmarks focus on determining whether the generated code meets specific functional requirements. EvalPlus [121] addresses the limitations of traditional benchmarks by generating a larger set of test cases through automatic test input generation, significantly expanding the scope of testing. Beyond isolated function synthesis, other benchmarks such as SWE-bench [81] focus on real-world software engineering tasks, where LLMs must resolve issues in complex codebases involving multiple files and components. SWE-bench evaluates LLMs by providing them with GitHub issues and corresponding codebases, requiring models to generate code patches that resolve the issues while passing the associated test cases.

Our work differs from these approaches by emphasizing the integration of real-time feedback and iterative evaluation during the code optimization process. While traditional benchmarks focus on evaluating functional correctness through static test cases, our system, MACEDON, offers a dynamic, user-centered approach, allowing programmers to evaluate their code's state across multiple dimensions and refine it interactively with the support of LLM-driven recommendations. This approach bridges the gap between one-off code

generation and the ongoing process of code refinement, offering a more practical solution for real-world programming tasks.

6.2.3 Positioning Relative to Traditional and LLM-based Tools

As Section 2.3.4 discusses, performance tools (e.g., HPCToolkit, Speedoo) and static refactoring tools (e.g., ESLint, JDeodorant) operate as separate analysis phases rather than providing real-time feedback. LLM-based assistants focus on code generation; programmers still need visibility into their code's state across multiple dimensions before crafting optimization prompts. Unlike search-based optimization [55] or task decomposition [217], MACEDON combines real-time, multi-dimensional evaluation (clarity, documentation, redundancy, runtime, memory) with interactive optimization. It provides immediate feedback during programming, uses LLM capabilities for context-aware recommendations, and integrates evaluation and optimization in one interface.

6.3 Design Process & Goals

We conducted an iterative user-centered design process to create MACEDON. The design process included three key stages: 1) *Understanding & Ideation*—involving an interview study with experienced programmers to uncover challenges and strategies in code optimization using LLM-driven tools; 2) *Prototype & Walkthrough*—the design and development of MACEDON, informed by the insights gained, followed by a cognitive walkthrough for feedback and iterative refinements; 3) *Deploy & Evaluate*—a user study to assess how programmers interact with the system and its perceived usefulness in streamlining code optimization. In this section, we describe the first stage of our design process, outlining the strategies and design goals that guided the development of MACEDON.

6.3.1 Interview Process

We recruited six participants (4 men, 2 women; self-reported gender; ages $20-27, M = 23.5, SD = 1.2$) with different levels of programming experiences through purposive sampling [46] for our interviews. Our goal was to engage participants who were both experienced in programming and familiar with LLM-driven code optimization tools. During the recruitment process, we conducted a pre-test survey to screen for eligibility, measuring programming experience on a 5-point scale [1: very inexperienced; 5: very experienced],

years of programming experience, and self-reported familiarity with code optimization tools driven by LLMs. Three participants are novice programmers with only around one year of programming experience while the other two have at least five years.

They were all familiar with programming (score $M = 4.17, SD = 0.41$), and regularly used LLM for code optimization purpose ($M = 7.5, SD = 2.10$ times/week). Participants gave consent and were compensated 20 CAD for a 60-minute study session.

Before the study, participants were asked to share their recent examples of ChatGPT usage for optimizing the code to nudge them to reflect their optimization strategies with LLM-code optimization tools. During the session, we interviewed participants to explore their challenges in understanding and evaluating the state of their code, translating them to optimize the code further, and their strategies for addressing these challenges and their needs. All interviews were audio-recorded and transcribed. We then performed thematic analysis [180] using both inductive and deductive approaches. Based on our analysis, we identified and categorized key themes and strategies employed by participants. Any disagreements were resolved through discussion, leading to final themes after a second iteration of analysis.

6.3.2 Interview Results

Here, we present our findings on the workflows that participants adopted during the code optimization process and the challenges they encountered.

6.3.2.1 Multi-Dimensional Code Evaluation

We observed that participants' approach to optimizing code involved two key aspects. First, programmers needed to assess the current state of their code across multiple dimensions, such as clarity, efficiency, and readability, which was crucial to determine "where the code falls short and what needs to be improved." [P2] Second, they had to explore how to implement optimizations in a way that addressed those specific shortcomings. However, participants encountered challenges with the lack of transparency in optimization recommendations, which made it difficult to understand how certain changes impacted their code. P4 noted, "It's hard to know if the optimization is improving performance or just adding complexity". To alleviate cognitive load, every participant adopted a similar strategy to break down the optimization process into smaller, manageable steps.

Six participants suggested that having a structured approach to multiple dimensions would make the optimization process more efficient. P4 emphasized, "I usually start with

time efficiency, then clarity and documentation to ensure everything is readable and well-organized." Additionally, two participants mentioned that they would easily overlook space usage and redundant code, which could affect others' ability to read the code efficiently. For example, P2 noted, "I don't always think about space efficiency unless flagged." P6 added, "It's only after someone struggles with reading my code that I realize how important reducing redundancy is." Thus, a multi-dimensional code evaluation approach is needed to optimize code in a systematic and organized manner.

6.3.2.2 Facilitating Direct Code Segment Optimization

Participants expressed challenges in managing large amounts of code, leading to a sense of control loss over the optimization process, where participants expressed the desire to "optimize the code dimension by dimension" (P2). Some (4/6) participants expressed frustration with the system applying changes across the entire code at once. Participants preferred direct manipulation of individual code segments based on specific priorities, such as time performance or clarity improvements. P5 explained, "I prefer to fix one section at a time; it helps me see exactly what's being changed and why." Participants (4/6) reported frequently using an alternative strategy where they optimized self-contained code segments independently before integrating them into the broader codebase. P4 noted, "I prefer optimizing each part separately and then merging them so I can see the overall improvements gradually." Another common approach involved selecting and optimizing specific segments of code based on previous runs, enabling programmers to refine code in targeted areas without affecting other sections. However, P5 also outlined this tedious process of preserving and comparing the newly optimized code and original code, "I have to keep track of which parts are optimized and ensure they don't conflict with the rest of the code."

6.3.2.3 Real-Time Feedback for Iterative Code Optimization

All participants expressed frustration with the constant switching between evaluating their code and applying optimizations, which often led to cognitive overload. P2 shared, "Sometimes I need to stop and evaluate if the optimization worked; it breaks my flow and makes the whole process slower." P5 agreed and mentioned that it was"frustrating to go back and forth between evaluation and deciding whether to make changes." However, we observed that participants preferred real-time feedback during the process, which helps them quickly verify "whether the code change is accurate" (P4). Some participants even described the benefits of iterative code optimization for each small code segment. P4 noted, "I often focus on the areas that require the most attention first and do iterative prompting until it

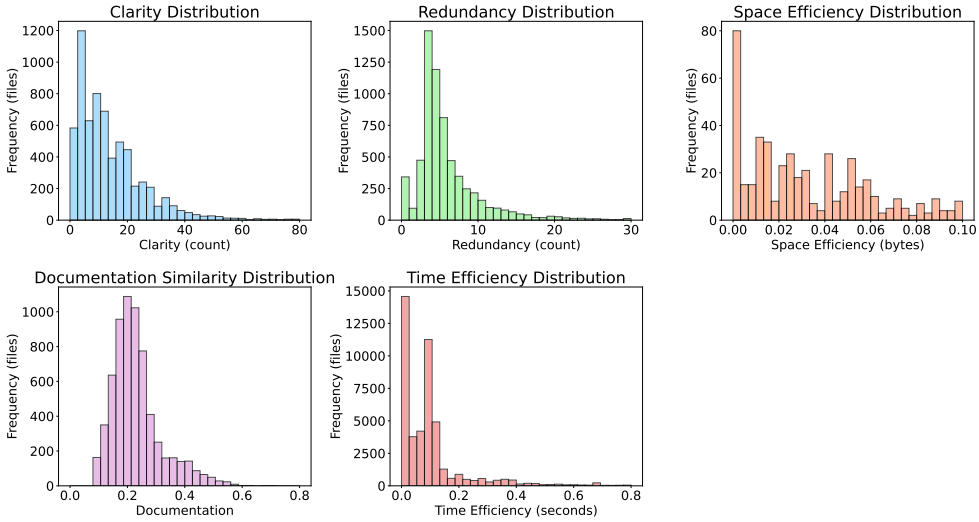


Figure 6.1: Distributions of key metrics: clarity, redundancy, time efficiency, space efficiency, and documentation in PIE dataset.

works." The iterative method of refining individual sections allowed participants to remain in control of the process, avoiding broad changes that might affect the entire codebase and ensuring each adjustment was aligned with their optimization goals.

6.3.3 Design Guidelines

Based on the interview results, we derived the following design guidelines to drive the development of MACEDON.

DG1: Providing Multi-Dimensional Feedback for Code Evaluation A lack of comprehensive evaluation across different metrics can prevent programmers from fully understanding the state of their code. The system should offer multi-dimensional feedback, such as performance, readability, and clarity, to help users better understand the strengths and weaknesses of their code. This evaluation should be presented in a structured and organized manner, allowing programmers to externalize their thought processes and refine specific areas of the code based on the feedback. Flexible feedback options should reflect the various dimensions programmers need for code optimization.

DG2: Facilitating Direct Code Segment Optimization Programmers often feel a loss of control when working with large amounts of code that require detailed optimization.

The system should support direct manipulation of specific code segments, enabling users to select and modify areas for improvement based on their priorities. It should also allow programmers to reorganize or refine their code incrementally, providing the ability to test and apply optimization recommendations to one segment at a time, instead of overwhelming them with global code changes.

DG3: Integrating Real-Time and Interactive Code Evaluation with Optimization Suggestions Programmers often face cognitive overload due to the constant need to switch between code evaluation and applying optimizations. The system should integrate real-time feedback with iterative suggestions, allowing programmers to continuously assess their code and apply optimizations without disrupting their workflow. By providing timely, contextual feedback during the optimization process, the system can guide users in refining their code in a smooth, iterative manner.

6.4 Define Code Evaluation Method

The insights of our formative study led to the investigation of our **RQ1** regarding evaluation dimensions and optimization strategies that programmers normally use. As indicated by **DG1**, we decided to evaluate the C++ code on five key metrics: redundancy, documentation, clarity, time efficiency, and space efficiency.

6.4.1 Data Collection

We utilize the Performance-Improving Edits (PIE) dataset [169] containing human-programmer optimizations from competitive programming tasks in CodeNet [156]. We selected C++ as our target language due to its prevalence in performance-critical applications and compatibility with benchmarking tools like the Gem5 simulator. C++ submissions typically emphasize fine-grained optimizations, making them ideal for analyzing time and space efficiency metrics.

To ensure data quality, we filtered submissions to include only C++ programs that passed all test cases and met runtime constraints. Each program was paired with its performance-improved version, enabling comparative analysis. We used the Gem5 simulator to normalize execution times, eliminating inconsistencies in raw runtime data. The resulting dataset contains 77,967 pairs of functionally correct programs across diverse problem domains.

6.4.2 Data Analysis and Results

6.4.2.1 High-Quality Code Characteristics

We began the analysis by describing the general characteristics of these C++ codes and deciding what parameters needed to be used in a five-dimensional framework. Programs in the PIE dataset demonstrate strong performance across all five optimization dimensions (Figure 6.1). Most programs exhibit high clarity (fewer than 10 issues per file) and low redundancy (fewer than 5 redundant constructs), indicating well-structured and efficient code. Space and time efficiency metrics cluster around optimal values, while documentation scores fall within moderate ranges (0.1–0.3).

6.4.2.2 Multi-Dimensional Code Quality Analysis

Table 6.1 presents our analysis of five key metrics across the PIE dataset. The results reveal distinct patterns:

Clarity: 58.65% of files contain fewer than 10 clarity-related issues, indicating generally well-structured code.

Redundancy: 32.36% of files show moderate to high levels of redundant constructs, suggesting optimization opportunities.

Space Efficiency: 19.19% of files consume significantly more memory than average, highlighting areas for memory optimization.

Documentation: 6.30% of files demonstrate high semantic alignment between code and documentation, while 5.60% show minimal documentation efforts.

Time Efficiency: 1.41% of files achieve exceptional performance, representing solutions that excel across all metrics.

6.4.2.3 Evaluation Strategies Align with the evaluation framework

Inspired by recent prompting strategies [223] and offline reinforcement learning(offline-rl) techniques [29], we introduced performance tags to evaluate C++ programs. This tagging scheme enables us to track how programmers adjust their optimization strategies over time. For time efficiency, we assigned tags by associating each "fast" program with an indicator of its optimal achievable performance across all solutions in the dataset. Specifically, the tag represents how close a program is to peak performance on a binned scale from 1 to 10, with

Metric	Formula	Evaluation Method	Examples (Original → Optimized)
Time Efficiency	$Score = \lceil 10 \cdot (1 - perc(T_{exec})) \rceil$	Algorithmic complexity and runtime behavior	`pow(x, 2)` → `x * x`
Space Efficiency	$Score = \lceil 10 \cdot (1 - perc(M_{usage})) \rceil$	Memory footprint and data structure usage	`vector<vector<int>> mat(n, vector<int>(n, 0))` → `vector<int> mat(n * n, 0)`
Clarity	$Score = \left\lceil 10 \cdot \left(1 - \sum_i w_i \, perc(f_i)\right) \right\rceil$	Magic literals, naming consistency, and statement length	`int a = 0;` → `int sum = 0;`
		Descriptive names and magic number avoidance	`int f(int x);` → `int computeSquare(int x);`
Documentation	$Score = \lceil 10 \cdot perc(S_{sim}) \rceil$	Embedding-based semantic similarity between code and documentation	`// calculate result` → `// Computes total revenue from all entries`
Redundancy	$Score = \lceil 10 \cdot (1 - perc(Redundant_{lines})) \rceil$	Duplicate code and dead code detection	`for (...) print(x); for (...) print(x);` → `for (...) print(x);`

Table 6.1: Evaluation methods, formulas, and optimization examples for each dimension. Notation: $perc(\cdot)$ returns a training-set percentile rank in $[0, 1]$; T_{exec} is execution time; M_{usage} is peak memory usage; $Redundant_{lines}$ counts duplicate/dead lines. For clarity, f_i are style factors with weights w_i ($\sum_i w_i = 1$). For documentation, S_{sim} is the cosine similarity between a code segment and its accompanying natural-language documentation (e.g., CodeBERT embeddings). The ceiling operator $\lceil \cdot \rceil$ maps the $0-1$ value onto the common $1-10$ rubric (top 10% receive 10).

the top 10% of optimized solutions for a given task labelled as "10/10," the next 10% as "9/10," and so on. Similarly, for space efficiency, tags are assigned based on memory usage during execution, with the top 10% of most memory-efficient solutions labeled "10/10," the next 10% as "9/10," and so on. This reflects how efficiently a solution uses resources.

As shown in Table 6.1, for other metrics (clarity, redundancy, and documentation), which lack direct quantitative parameters, five members of the research team analyzed the code files to identify specific evaluation parameters within each category. For clarity, we assess magic literals, naming consistency, and statement length based on static analysis. For documentation, the distribution was calculated embedding- based semantic similarity between code and documentation. Redundancy detection identifies duplicate code constructs and unnecessary operations(e.g., code constructs redundancy, dead code, and unused code). These thresholds follow established methods in code quality evaluation [34, 168], ensuring our scoring framework aligns with industry standards. Each factor is converted to its percentile rank in the training set, with lower-is-better measures (runtime, memory, redundant lines) inverted (see Table 6.1 for formulas and before/after examples). These percentiles are combined with predefined weights and then rounded up to a 1-to-10 scale by computing $\lceil 10 \times perc \rceil$.

The final step applies the same ten-group scale: the top 10% for any metric score 10, the next 10% score 9, and so forth. This common rubric keeps all five metrics on a comparable 1–10 scale and aligns them with industry standards.

6.5 MACEDON: Design and Implementation

The results of our formative study and data analysis confirmed the need for multi-dimensional code evaluation and optimization. This led us to design and develop MACEDON, addressing the first part of **RQ2**, which aimed at assisting novice programmers in improving their code efficiency, readability, and maintainability via the multi-dimensional framework.

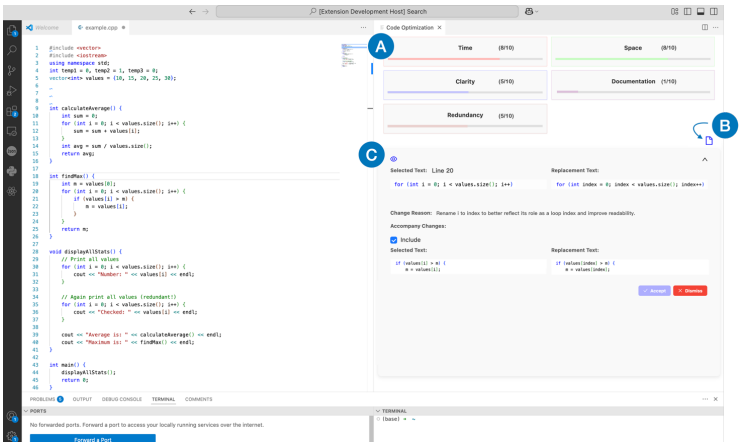


Figure 6.2: MACEDON is an extension of the Visual Studio Code. It provides three basic dashboards: (A) Evaluation Dashboard provides the current score of their code in five dimensions(clarity, documentation, redundancy, time efficiency, and space efficiency); (B) a button for user to trigger Optimization Dashboard; (C) Optimization Dashboard provides a prioritized list of actionable suggestions based on the real-time evaluation results in each dimension.

6.5.1 System Architecture

The MACEDON system consists of two main components: 1) a client-side UI, implemented as a Visual Studio Code (VS Code) extension using TypeScript, and 2) a server-side backend, developed in Python and Flask.

The client-side program is responsible for rendering the UI and tracking user interactions with the code editor. The client-side component is responsible for rendering the optimization dashboard and tracking user interactions within the code editor. When a user modifies a code segment, MACEDON sends the updated content to the server-side program via HTTP requests for real-time evaluation.

The server-side processes the code using a multi-dimensional analysis framework that evaluates performance, readability, redundancy, and documentation quality. For performance optimization, the server utilizes benchmark-based profiling and heuristic-guided transformations to suggest improvements. For readability and redundancy detection, MACEDON employs a static analysis pipeline to identify redundant constructs and enhance clarity. Additionally, for documentation generation, MACEDON integrates natural language processing (NLP) techniques, extracting context from the code and generating inline expla-

nations. The system also supports cascading updates, ensuring that optimization changes propagate across dependent code segments for consistency.

6.5.2 User Interface Design

Figure 6.2 shows the UI of MACEDON as a VS Code Extension, which comprises two main UI components.

The Evaluation Dashboard presents a multi-dimensional analysis of the code in real time, covering key dimensions including time efficiency, space efficiency, clarity, redundancy, and documentation (Figure 6.2.A). Each dimension is visualized with a unique color (red for time efficiency, purple for documentation, blue for clarity, green for space efficiency, and pink for redundancy), enabling users to quickly identify which aspects of the code require attention.

The Optimization Dashboard provides a prioritized list of actionable suggestions based on the real-time evaluation results (Figure 6.2.C). Suggestions are grouped by dimension and ranked by impact, offering users a clear entry point for making improvements. When a user clicks on a suggestion, an optimization panel appears, showing the original code, the proposed change, and the rationale behind it (Figure 6.2.C). Users can accept changes with a single click or further customize them. For example, they may rename a variable to enhance clarity or replace a nested loop for improved time efficiency.

6.5.3 Approach for Multi-Dimensional Code Evaluation and Optimization

6.5.3.1 Code Evaluation

Our formative study suggests that the system should support structured, multi-dimensional code evaluation, covering key aspects such as performance, readability, redundancy, space efficiency, and documentation quality to provide programmers with a comprehensive understanding of their code.

Some types of code evaluation can be directly derived from static analysis, allowing for automated assessments. For example, clarity can be analyzed using Clang-based static analysis, identifying areas where variable naming, indentation, or code structure could be improved for better readability. Similarly, time efficiency and space efficiency can be

assessed through runtime profiling, where CPU execution time and memory consumption are monitored to detect performance bottlenecks and inefficient memory usage.

Other types, including redundancy and documentation quality, require deep learning-based approaches to capture more abstract patterns. MACEDON leverages embedding-based similarity analysis to detect redundant code structures by comparing functionally similar but structurally different code blocks. For documentation evaluation, MACEDON uses a semantic similarity model to analyze the alignment between existing code comments and the corresponding implementation, identifying areas where documentation is incomplete or inconsistent. Through this hybrid evaluation approach, MACEDON provides targeted optimization recommendations while maintaining a balance between automated insights and user-driven refinements.

Deep Learning Approach: We propose a deep learning approach to detect code redundancy and evaluate documentation quality. For redundancy detection, our method leverages a hybrid embedding model that captures both syntax and semantics by processing Abstract Syntax Trees (ASTs) alongside tokenized code. This multi-level analysis enables the detection of identical fragments as well as functionally similar yet structurally diverse code. We trained a Graph2Seq [215] model on 8,500 manually- and automatically-labeled code pairs using contrastive loss over 120 epochs on an H100 GPU cluster. Our approach achieved an F1-score of 87.3%, outperforming traditional token-based (72.5%, e.g., CCFinder [86]) and AST-based (81.2%, e.g., Deckard [79]) methods.

For documentation quality evaluation, a transformer-based model (e.g., CodeBERT) was fine-tuned on 12,000 code-comment pairs, including synthetic paraphrased data. This model assesses whether inline comments and docstrings adequately explain the corresponding code. Evaluated with BLEU, METEOR, and BERTScore metrics, the model achieved a BERTScore of 0.84, demonstrating its ability to capture semantic relevance even when lexical differences exist.

A qualitative analysis on 50 random code snippets indicated that while our model occasionally overestimates redundancy in boilerplate code and under-scores concise but sufficient documentation, it generally distinguishes between necessary repetitions and truly redundant patterns, as well as highlights missing explanations for complex logic.

Statistic Analysis Approach: Our formative study shows that key aspects of code quality—clarity, time complexity, and space complexity—can be effectively evaluated using static analysis without deep learning.

We use Clang-based static analysis to identify readability issues such as inconsistent naming, deeply nested loops, hardcoded values, ambiguous function names, and poor in-

dentation. Clang's AST parsing also detects complex control structures and suggests refactoring strategies.

Time complexity is estimated by profiling loop structures and function calls. By analyzing the AST, our system extracts nested loops, recursion, and dependencies to approximate computational complexity based on known algorithm patterns. A cost-modeling approach further estimates execution time from static operation counts, eliminating the need for runtime profiling.

Space complexity is assessed by examining memory allocations. Our analysis tracks variable sizes, pointer usage, and dynamic allocations through Clang's type analysis, distinguishing between heap and stack usage. Redundant copying and memory leaks are flagged, with recommendations for optimizations such as pass-by-reference or in-place modifications.

This integrated static analysis framework offers a lightweight yet effective method for evaluating code quality and can be extended to other programming languages by adapting AST parsing and memory tracking techniques.

6.5.3.2 Code Optimization

We fine-tuned a transformer-based model, Qwen-Coder[1], for multi-dimensional code optimization using a targeted dataset rather than large-scale data. Instead of relying on intuitive few-shot in-context learning, we adopt a prefix-tuning supervised approach by clearly stating the desired optimization dimension (e.g., "Clarity:", "TimeEfficiency:") in the prompt.

For data preparation, we curated approximately 500–800 code edits from public GitHub commits. Each commit was filtered using heuristic keywords (e.g., "refactor," "speed up," "optimize memory," "remove duplication," "improve docs") and manually validated to ensure the edits reflected genuine improvements across five dimensions: clarity, time efficiency, space efficiency, redundancy, and documentation.

We fine-tuned a single multi-task model over 60 epochs (batch size 16) with early stopping on a single H100 GPU. The unified model learns dimension-aware code transformations, allowing users to steer the optimization by specifying the target dimension in the prompt during inference. This design supports local, iterative deployment, where developers receive real-time, tailored code improvement suggestions.

[1] https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct

Evaluations were conducted using both quantitative and qualitative metrics. For clarity, the Maintainability Index (MI) increased from 68.2 to 81.7, while Cyclomatic Complexity (CC) dropped from 4.6 to 3.1 on a test set of 100 samples. Time efficiency improvements included an average 1.82× speedup, with 63% of cases showing reduced algorithmic complexity (e.g., from $O(n^2)$ to $O(n \log n)$). Memory profiling revealed an average reduction of 27.4% in memory usage. Redundancy metrics showed a 42.5% decrease in redundant code lines, and documentation quality, measured using a fine-tuned CodeBERTScore model, improved to a BERTScore of 0.842, outperforming standard models.

The MACEDON dashboard integrates these metrics—performance, clarity, and redundancy—allowing programmers to visualize and directly manipulate their code quality evaluations.

6.5.3.3 Facilitating Code-Optimization Correspondence

MACEDON offers features that enable programmers to navigate various code optimization suggestions while maintaining alignment between the code and its corresponding feedback blocks (**DG3**).

Showing Corresponding Code. The code editor highlights relevant code segments based on the selected optimization suggestion. Each optimization dimension, such as performance, readability, or space efficiency, is represented with a distinct color. For example, clarity issues may be highlighted in blue, while space efficiency suggestions appear in purple. This ensures that programmers can quickly identify which part of the code relates to a specific optimization, with other code sections folded to reduce visual clutter.

Dependency Highlight and Updates. When code segments are modified, Macedon triggers related updates across dependent code sections, ensuring consistency. These dependent code segments are highlighted in distinct colors based on the type of update, with opacity indicating the relevance of each segment to the original suggestion. For example, renaming a variable will not only update that specific segment but will also highlight related usages throughout the code, with clearer visibility on more relevant instances. This helps the programmer track how a single change impacts the rest of the code, maintaining consistency across the entire codebase.

6.6 User Evaluation

To further investigate **RQ2** regarding MACEDON's effectiveness, we conducted a within-subject controlled experiment which aimed at understanding (1) how well it facilitates code optimization across multiple dimensions and (2) how programmers perceive the multi-dimensional optimization strategy.

6.6.1 Participants

We recruited 24 novice programmers (16 men, 8 women; self-reported gender; $M = 20$, SD = 3.54) to participate an in-lab user study evaluating the MACEDON. All participants had less than one year of programming experience and C++ proficiency scores of 2 or lower on a 5-point scale ($M = 1.42$, SD = 0.51). To ensure familiarity with LLM-based code assistants (e.g., GitHub Copilot), we screened participants using a self-assessed 5-point Likert scale measuring familiarity ($M = 4$, SD = 0.74) and required prior experience with code generation using such tools, which participants reported using regularly ($M = 8$ times/week, SD = 2.56). We used a snowball sampling approach to recruit participants, where we sent recruitment messages to friends, colleagues, and various university mailing lists. We then asked participants to refer their friends and colleagues. Following the tests, we conducted semi-structured interviews and distributed questionnaires to gather user feedback.

6.6.2 Study Protocol

A within-subject design was employed. Participants' task was to evaluate and optimize a C++ code given by us. The task focused on evaluating MACEDON's core feature—multi-dimensional code optimization. Without being explicitly provided with guidance on editing, participants were asked to improve their code until all five quality scores (clarity, documentation, redundancy, time efficiency, and space efficiency) reached ≥ 8. This setup avoids bias and preserves open-ended exploration while ensuring comprehensive evaluation across all dimensions. The independent variable was the optimization tool used (MACEDON vs. baseline). Each participant completed the task with both tools in counterbalanced order. To simulate a realistic development context, participants were told that they were optimizing the code for the purpose of sharing code for their professor or interviewer who needs to evaluate their code and give them scores. Each participant is asked to finish two

sessions, one with the MACEDON support and one with Copilot. We chose GitHub Copilot as the baseline because it is the most widely adopted AI code assistant among our target population. This comparison lets us evaluate whether multi-dimensional feedback provides benefits beyond what a general-purpose code assistant offers. We prepared two draft C++ codes, one for each session. The two experimental C++ code are adapted from Github. The two C++ codes have a similar length and a similar level of difficulty. To counterbalance the order effect, we randomized the order of the Copilot condition and the MACEDON condition for each participant, so some participants encountered MACEDON in their first session, and some others experienced it in their second session. To reduce bias from identifying the baseline tool by name, we referred to both conditions as "Tool A" and "Tool B" during the study. Participants were debriefed about the tool identities only after completing both conditions. This way, we could investigate the capabilities of MACEDON from multiple perspectives more thoroughly.

Each session was limited to 30 minutes to finish one session. We conducted three pilot runs in which all pilot participants were able to complete the tasks within 20 minutes, with MACEDON or with Copilot. Prior to the MACEDON condition session, participants were given a 2-minute quick demo of MACEDON 's interface and features. All study sessions were conducted remotely over a video conferencing tool. We asked participants to share their screens and we video recorded the entire session with their permission. After completing both sessions, we conducted a post-study semi-structured interview to gather qualitative feedback. Participants were asked questions such as "How did MACEDON 's evaluation and optimization compare with the baseline tool in helping you improve code?" and "Which optimization dimensions did you find most useful or challenging?" Additionally, participants were encouraged to share their reflections on system usability, tell their stories, and experience outside these structured questions. The interview sections of the video recordings were transcripted into text.

6.6.3 Data Collection and Measurement

We collected three types of data sources: the screen recordings and observational notes for each session, the final optimized code artifacts from each session, and the post-task questionnaire responses and interview transcripts.

Our first group of measurements focuses on behavioral data extracted from session recordings. For each session, we recorded the task completion time (in seconds). In the condition with MACEDON support, we also counted: how many times a participant triggered MACEDON (e.g., clicked on a specific optimization tag); how many times the participant

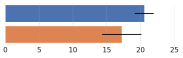
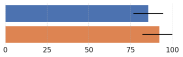
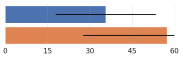
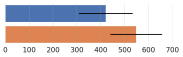
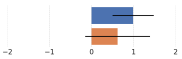
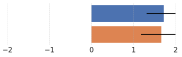
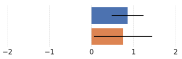
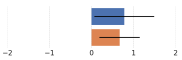
	Condition	M	SD	
Number of Line changes	MACEDON	20.54	1.41	
	Copilot	17.21	2.90	
Number of token changed by tools	MACEDON	85.57	8.89	
	Copilot	92.26	10.24	
Number of tokens human changes	MACEDON	35.62	17.81	
	Copilot	57.32	29.64	
Task Completion Time (secs)	MACEDON	421.17	112.45	
	Copilot	548.33	108.61	
Satisfaction with the Final C++ Code (-2 to 2)	MACEDON	0.99	0.49	
	Copilot	0.62	0.77	
Expert Rating: Readability (-2 to 2)	MACEDON	1.72	0.41	
	Copilot	1.66	0.48	
Expert Rating: Maintainability (-2 to 2)	MACEDON	0.86	0.38	
	Copilot	0.75	0.69	
Expert Rating: Performance (-2 to 2)	MACEDON	0.78	0.71	
	Copilot	0.67	0.48	

Table 6.2: Performance data in two conditions (M: mean, SD: standard deviation): the task completion time (secs), participants' satisfaction with the final code artifact (from -2 to 2), code quality, number of line, and number of tokens. In particular, participants spent less time to complete the task in the MACEDON condition than the Copilot condition ($p = .001$); participants were more satisfied with the final Code in the MACEDON condition than the Copilot condition ($p = .001$).

adopted a suggestion generated by MACEDON (code lines directly modified using system suggestions); how many times the participant ignored the system-generated recommendation and manually edited the code (manual optimizations); and how many times the final version of the C++ code is co-created by users and MACEDON.

Second, to evaluate the quality of the final C++ Code artifact, we define our second group of measurements by counting: the number of lines changed and the total number of tokens changed by MACEDON or Copilot, the total number of tokens changed by humans, as these three are indicators of the quantity and effort each participant spent on the C++ Code. In addition, we asked participants to rate their own satisfaction with the final optimized code for each condition using a 5-point Likert scale (-2 to 2). To objectively assess

optimization quality, we asked two C++ experts (C++ Programming Years: $M = 10.1$ years, SD = 2.34) to rate the C++ code-level quality using a 3-dimensional rubric measuring improvements in readability, maintainability, and performance. Each dimension was scored on a scale from -2 to 2. Two experts iteratively discussed and evaluated the C++ code until the independent ratings achieved an agreement (Krippendorff's alpha: $\alpha = 0.76$). The result of this analysis is reported in Table 6.2.

Finally, we asked the participants to finish a post-experiment survey (5-point Likert Scale, -2 as strongly disagree to 2 as strongly agree) to assess participant perceptions of MACEDON across multiple factors, including usability, accuracy, trust, satisfaction, and adoption propensity based on the instrument by Weisz et al. [206]. These survey results are summarized in Figure 6.3. Pairwise comparisons showed that MACEDON received significantly higher ratings than the baseline on usefulness ($p < .01$), trust ($p < .05$), and satisfaction ($p < .01$). Differences in accuracy and adoption propensity were not significant. For the interview transcripts, three researchers of this project conducted an iterative open coding method to get the code, theme, and representative quotes as another group of data. They each independently coded a subset of interview transcripts and discussed the codes together. Through iterative discussion and re-coding, we applied the codes and themes to their assigned code samples. Some examples of the identified themes include: pros and cons of MACEDON, experiences with specific optimization dimensions, preference of the multi-dimensional code evaluation and optimization approaches, expectations for future adoption, and suggestions for system design improvement. These qualitative insights are reported with quantitative findings to provide a comprehensive view of user experience with MACEDON.

6.6.4 Results

In this section, we present the results of our user study, focusing on how MACEDON improves participants' performance on the task, how participants perceived the multi-dimensional code evaluation and optimization methods in MACEDON, and how participants described the usability and effectiveness of MACEDON.

6.6.4.1 MACEDON Supports Participants to Easily Optimize the Code

Our study shows that MACEDON improved participants' code optimization performance by reducing task completion time and enhancing the satisfaction of the final code.

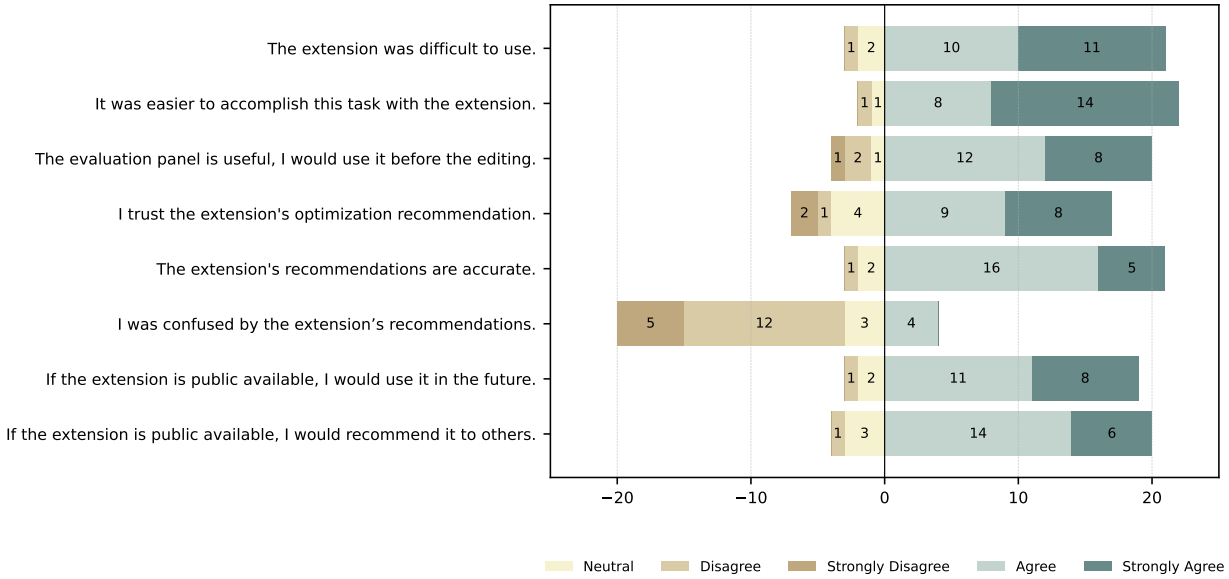


Figure 6.3: Results of the post-task questionnaire. The numbers indicate the counts of participants on corresponding scales.

We conducted a two-way repeated measures ANOVA to examine the effect of tool usage (with MACEDON or with Copilot) on task completion time. The two factors were tool condition (MACEDON vs. baseline) and optimization dimension (clarity, redundancy, documentation, time efficiency, space efficiency). As shown in Table 6.2, participants completed the optimization tasks significantly faster with MACEDON ($M = 421.17$, $SD = 112.45$) compared to the condition using Copilot ($M = 548.33$, $SD = 108.61$). Besides, there was not significant effect of codes was found on task completion time.

The post-experiment survey result further supported our findings. Most participants agreed that MACEDON helped them finish the task with MACEDON's help (23 out of 24 rated agree or higher), as shown in the Figure 6.3. Participants also found the code evaluation and were accurate (19 out of 24). Participants noted that the integration into VS Code and the dimension-tagged interface enabled them to optimize specific code segments directly, thus it was easier for them to start optimizing the code: "It's not just about fixing bugs, it shows me where things can improve — like this loop can be faster." (P4)

6.6.4.2 Multi-dimensional Code Evaluation in MACEDON Yields Better Quality of Code and Improves Accuracy and Readability

Through coding the video recordings for two sessions, we were able to examine the following questions: While the MACEDON was available, how did the participants use it, especially multi-dimensional code evaluation during code optimization? Did they check the evaluation scores from MACEDON when optimizing the code? Did they actually use those code recommendations under different dimensions when optimizing the code? Did they adopt them directly, or did they revise them for their specific needs?

Based on our findings, when MACEDON was available, most participants (83.3%) viewed evaluation scores at first by clicking on the MACEDON extension in the Visual Studio setting. Most participants (91.7%) not only checked these scores, but also opened the recommendation panel linked to the evaluated dimension generated by MACEDON.

We also found that each participant triggered recommendations ($M = 14.2$, $SD = 4.1$), directly adopted suggestions without modification ($M = 8.9$, $SD = 3.6$), ignored those deemed unhelpful ($M = 3.2$, $SD = 2.5$), and co-created solutions by modifying suggestions before accepting them ($M = 2.1$, $SD = 1.4$). Overall, 62.5% of final edits were adopted directly from MACEDON's suggestions, demonstrating strong reliance on system-generated feedback. As P7 described, "It gave me a starting point, like changing a loop, but then I tweaked the logic to better fit my approach." An interesting pattern emerged regarding participants' optimization workflows. While individual preferences varied, most participants followed a structured sequence when applying improvements. Specifically, 15 out of 24 participants began with either redundancy or clarity, followed by space and time efficiency, and ended with documentation. The three most common sequences were: (1) redundancy → clarity → space efficiency → time efficiency → documentation (9/24), (2) clarity → redundancy → space efficiency → time efficiency → documentation (6/24), and (3) time efficiency → space efficiency → redundancy → clarity → documentation (6/24). (4) space efficiency → time efficiency → redundancy → clarity → documentation (6/24).

In order to explore the differences between workflows that start with redundancy/clarity versus those that start with space/time efficiency, start from documentation, we conducted a code-level expert rating along the dimension of readability, maintainability, and performance. Each expert needs to rate 48 code samples generated from Copilot and MACEDON. Specifically, readability is the extent of which a human reader can understand the purpose, control flow, and structure of the code. Maintainability describes how easy it is to understand, modify, extend, and fix the code over time. Performance describes the code's efficiency in terms of runtime speed and memory usage. We also calculated the token count of the code. We performed a one-way ANOVA to examine the differences among the three

groups. As shown in Table 6.2, participants who optimize the code starting with redundancy or clarity yielded significantly higher ratings in both readability ($F = 8.48, p < 0.001$) and maintainability ($F = 1.25, p < 0.001$), while for performance, there are no significant differences across the three groups. Our posthoc analysis also suggested that no significant differences were found between code optimized starting from space/time efficiency and documentation only along all dimensions (including readability, maintainability, and performance).

6.6.4.3 MACEDON increases participants' satisfaction

The post-task questionnaire revealed that participants were significantly more satisfied with the code optimization experience when using MACEDON condition compared to the condition with Copilot ($F = 3.46, p < .001$ (Figure 6.3). P3 mentioned that the dimension-based optimization recommendation helped them feel more in control: "I have an overview of what to change. This actually gave me a checklist, which made things easier."

Some participants also highlighted that MACEDON reduced their thinking load during optimization. Instead of needing to decide where to start or what to fix first, MACEDON reminds them to edit the code that they may ignore. "It's like having a to-do list built into the code. I didn't feel lost." (P8)

Moreover, participants believed that MACEDON can help them form a better optimizing code practice in the long term: "It is useful to remind me which aspects of the code I usually overlook in a timely manner." Even though the quality scores for final code artifacts rated by two expert reviewers across four dimensions (clarity, time efficiency, redundancy, documentation) were not significantly different between the MACEDON and Copilot. (Figure 6.3), participants(P11-14) reported a smoother and more confident experience using MACEDON. As P11 noted, "I've had a much better experience with MACEDON. Most of the time, its optimization suggestions are accurate. What I really like is that it can update all the related code in one go with just a click. Even though sometimes I didn't fully agree with one recommendation, it still made me think, which I found really helpful." In some cases, we believe that while MACEDON improves programmers' productivity during code evaluation and optimization, it may not always produce code that aligns perfectly with individual coding styles or domain-specific practices. Thus, programmers may still prefer to revise or refine the naming conventions, formatting, and logic structure to ensure the final code fits their own standards and the broader context of their project.

In summary, while our experiment show that MACEDON improves the perceived usability and satisfaction of the code optimization process. By offering real-time, multi-

dimensional code evaluation and optimization, MACEDON also enhances productivity and supports a more reflective and satisfying development workflow.

6.6.4.4 The multi-dimensional optimization strategies in MACEDON are suitable for different user goals and expertise levels

In this section, we examine how participants perceived the different dimensions that MACEDON supports during code optimization. During the post-experiment interviews, we introduced the design behind each dimension and asked participants which dimensions they preferred, and in what contexts they found them most useful.

Participants commonly reported that the clarity and redundancy dimensions were useful entry points, especially when first encountering unfamiliar code. These suggestions helped them clean up structure and naming conventions, which improved overall readability. As P6 noted, "I always start with clarity or redundancy. If I don't understand or forgot my own code, performance won't matter."

The time efficiency and space efficiency dimensions were considered as more technical and goal-specific from participants. Several participants (P1-3, P7-9 P17-18) appreciated the performance profiling features, but only used them when optimizing for speed or handling large data. "I only look at time when I know performance matters. Otherwise I just skip it since sometimes I'm familiar with high-level algorithm and I'm not sure whether it is correct." (P8) Many participants prefer optimizing the code in clarity and redundancy first compared to simply improve the space and time efficiency. P10 mentioned, "I liked that I could ignore performance stuff at first and just focus on cleanup. Later, I switched to the time efficiency tab to improve it."

The documentation dimension had different reactions from our post-experiment interview. Some participants(P6, P22-23) said they rarely used it since they preferred to document in their own words. Others felt it was helpful to maintain consistency across their code: "It's a nice reminder, but sometimes I prefer to rewriting it in my way." (P2)

These findings suggest that different dimensions in MACEDON cater to different goals: clarity and redundancy help with code understanding and onboarding, while performance dimensions are more aligned with production-quality code. In particular, novice programmers benefited from a structure that guided them from readable to efficient code in a step-by-step way.

Participants (P11-13, P20-24) also highlighted that their optimization strategies depended on different scenarios. When writing code for themselves, they focused more on

clarity. When sharing or submitting the code, they paid more attention to performance and documentation. As P11 summarized, "I care if it works and is readable for my own stuff. I care about speed and explaining it better if I need to share with my friends or teachers."

These insights indicate that future versions of MACEDON could personalize workflows based on user preferences or experience level. For example, a scenario-aware mode could prioritize readability and clarity during individual or exploratory coding, while switching emphasis on performance and documentation when the code is intended for collaboration or submission.

6.6.4.5 Summary of the Results

In summary, our study shows that MACEDON significantly improves task efficiency and user satisfaction during code evaluation and optimization tasks. Participants actively engaged with the system's feedback in multiple dimensions. Expert evaluation data further suggest that MACEDON helped them produce higher-quality code across multiple dimensions. Overall, participants enjoyed using MACEDON and perceived MACEDON as a valuable assistant for both learning and real-world development workflows, particularly for tasks that require balancing performance and maintainability. We note that the controlled study recruited only novice programmers; the case studies with more experienced users suggest that the tool remains useful, but further studies with expert programmers are needed to confirm these findings generalize beyond novices.

6.7 Case Studies

To further examine how MACEDON generalizes to real-world usage scenarios (**RQ3**), we conducted an in-depth review of two programming cases by inviting two participants to use the system based on their real-life needs in two one-hour hands-on sessions. These two cases span diverse settings, including early-stage programming task and code interview programming. We intend to: (1) illustrate how MACEDON adapts to different user needs and workflows and (2) contextualize the MACEDON framework through practical examples.

6.7.1 Early-Stage Programming Task

To learn how MACEDON can be used in early-stage learning, we recruited a first-year undergraduate student (S1) who had completed only an introductory programming course. As part of their daily assignment for the programming course, they needed to ensure their code had great clarity and great time efficiency when submitting to the teacher, as they wanted to get a good score for this class. The example is shown in Figure 6.4.

Exploring Optimization Dimensions. S1 launched MACEDON from the Visual Studio Code extension interface and began reviewing their assignment code, a sorting algorithm that they had written in a previous lab. They were immediately drawn to the Optimization Dashboard, which highlighted areas of concern using color-coded scores across clarity, redundancy, time efficiency, space efficiency, and documentation.

S1 quickly noticed the code had low scores in clarity and redundancy. They clicked into these sections and received immediate feedback, including suggestions to rename variable names like `tmp1` and remove duplicated branches in conditionals. They commented: "I didn't think 'tmp1' was a problem, but now that I see it flagged, I agree it looks messy." They then accepted the renaming suggestion and applied a small logic refactor suggested by the tool.

They also explored the time efficiency section, which recommended replacing a nested loop with a more optimized standard library call. S1 was unfamiliar with the suggested function and used the built-in explanation panel to understand its purpose. "I've never used that function before, but the explanation helped. I can tell this will run faster for big inputs."

Customizing Optimization Suggestions and Learning Via Edits. Once S1 became comfortable with the interface, they began interacting more deeply with the suggestions offered by MACEDON. For each dimension, the system displayed not only suggested edits but also brief rationales and potential trade-offs. In the time efficiency category, MACEDON flagged a linear search loop and suggested using a set for faster lookup. S1 hesitated, and mentioned: "I've never used a set in C++... won't that make it more complicated?" The explanation tooltip clarified that using a set would improve performance for large inputs, and provided a short code example.

Motivated by this explanation, S1 attempted to rewrite the loop using the *std::unordered_set* structure. Although they initially struggled with the syntax, they used the integrated documentation feature within MACEDON to reference an example. After some trial and error, they got the revised code to compile and work correctly. "This is the first time I used a set. I probably wouldn't have tried it if the tool didn't suggest it." This moment was sig-

nificant—it illustrated how MACEDON not only suggested improvements but also acted as a learning scaffold, empowering the student to try out unfamiliar constructs.

S1 also customized a few suggestions that they felt were too generic. For example, when MACEDON proposed replacing a loop with a built-in function, S1 opted to preserve the loop but renamed variables and added comments for clarity. This show emerging confidence in balancing tool guidance with personal judgment. The ability to selectively adopt, adapt, or reject suggestions made MACEDON feel less like a rigid evaluator and more like a collaborative assistant.

Reflecting on Learning and Building Confidence. At the end of the session, we asked S1 to reflect on their overall experience with MACEDON. They emphasized that the most helpful part was seeing specific suggestions categorized by dimension: "It's different from getting a bunch of corrections at once. I can focus on one thing at a time—first make it cleaner, then think about speed." This multi-dimensional recommendation helped them develop a more systematic approach to refining code, which was previously absent from their workflow.

S1 also appreciated that MACEDON gave them the freedom to make the final call. They said: "It's not like the tool is telling me I'm wrong. It's more like: here's something you might not have thought about." They expressed a willingness to use MACEDON in future assignments, particularly for reviewing their code before submission. They also noted that seeing their own improvements logged in the history panel boosted their confidence: "It's cool to see all the changes I made. Feels like I'm actually getting better."

Finally, S1 mentioned that they would recommend MACEDON to their classmates, especially those who struggle with identifying what to improve in their code. They concluded: "We're all still figuring things out. Having something like this—it doesn't just fix your code, it shows you how to think about it differently." The case study demonstrates how MACEDON can serve not only as a productivity enhancer, but also as an educational ally, shaping how novice programmers learn to reason about quality in code.

6.7.2 Programming Evaluation During the Interview

To assess MACEDON's utility for experienced developers, we invited a senior software engineer (S2) who had 10+ years of experience in backend systems and frequently conducted technical interviews. As part of her work, she had to evaluate the C++ code written by interview candidates.

Rapid Multi-Dimensional Evaluation. S2 began by pasting a recent candidate's C++ submission into the Visual Studio and launched MACEDON to evaluate the code's

quality. He immediately noted the dimension-based feedback provided by the Optimization Dashboard. Color-coded bars surfaced low-performing dimensions, particularly redundancy and clarity. He appreciated that each suggestion was paired with a concise explanation and visual highlights in the editor: "This kind of feedback is exactly what I usually provide during interviews, but it's great to see it automatically structured like this."

He found the breakdown by dimension particularly useful in identifying overlooked issues. For example, MACEDON pointed out a duplicated loop construct and variable names like `res1` and `temp2` that reduced readability. He noted that "These are things I'd catch manually, but having the tool flag them saves time. It's like a second pair of eyes." He also mentioned that the time and space efficiency scores helped him quickly understand whether the solution was over-engineered or underperforming—without running the code. "Just looking at complexity isn't enough. These metrics back that up with context."

Comparing Candidate Submissions. Next, S2 opened a second candidate solution for the same problem to compare the two implementations side-by-side. He find the evaluation panel is very intuitive for him to directly know the performance of the current code in four dimensions. He can easily take the screenshot to compare the current candidate's submission to the previous submission by other candidates. He mentioned: "In interviews, I'm often reviewing 10+ candidates for the same problem. Being able to rank and compare by dimension is incredibly useful."

S2 used the Optimization Panel to review the candidate's code weakness, which could help him streamline hiring decisions, especially in situations where multiple reviewers assess the same submission. "This gives me a structured way to explain why one solution is stronger than another. It adds consistency to the evaluation process."

He also experimented with editing the weaker submission directly in MACEDON using its built-in suggestions. He found that small improvements in naming and redundant logic removal quickly raised the clarity score. "It's a good teaching tool too. I could use this during live interviews to show candidates how to improve their code."

Mentorship and Continuous Use. Finally, S2 reflected on how MACEDON could support not only hiring but also mentorship. He envisioned using it during onboarding to help junior engineers understand code quality standards within the team. "I could see using this in code reviews—not as a replacement, but to catch common issues and free me up to focus on architecture and logic."

He appreciated that MACEDON did not enforce a rigid style but provided adaptive, dimension-based guidance. "It doesn2019t tell you "this is wrong," it says "here's what could be clearer or faster," which fits well with how I coach junior devs." When asked whether he

would use MACEDON in the future, S2 responded affirmatively, especially for repetitive review tasks and mentoring sessions.

In summary, the case study shows that MACEDON offers tangible value for experienced engineers engaged in code review and technical interviews. Its structured evaluation, real-time guidance, and comparison tools can streamline both hiring decisions and educational feedback, supporting quality and efficiency across professional workflows.

6.8 Discussion

6.8.1 Automated Code Review Practices in Software Engineering

While automated code review systems in software engineering largely operate under the assumption of well-structured, production-ready codebases, our experiment with MACEDON reveals that professional programmers operate across a more diverse and context-sensitive spectrum of goals. Through our session with an expert software engineer, we observed that code review in real-world settings often requires balancing trade-offs between clarity, maintainability, and performance, none of which can be linearly optimized. For instance, as our case study and lab experiment show, participants frequently prioritized clarity and redundancy first before addressing performance dimensions, especially when reviewing code authored by junior developers. This mirrors findings in prior studies that emphasize the human judgment required in triaging automated suggestions [159] and customizing review based on code ownership and complexity [16].

In contrast to many existing tools that provide a flat list of suggestions or flag violations against static rulesets [24, 129], MACEDON's dimension-based feedback model encourages structured decision-making across clarity, redundancy, time efficiency, space efficiency, and documentation. Our quantitative analysis demonstrates that when users began optimization workflows with structural dimensions like clarity or redundancy, their final code artifacts achieved significantly higher accuracy and readability ratings, aligning with goals typically emphasized during onboarding and mentoring. This diverges from code review workflows that are primarily focused on defect detection and performance bugs [129, 181].

Furthermore, our observations resonate with recent discussions in the HCI and software engineering communities that advocate for more context-aware, human-centered design in AI-powered developer tools [138]. For example, research on mixed-initiative review systems suggests that trust and user control are critical to adoption [99]. MACEDON supports this

by allowing participants to selectively apply, edit, or ignore system suggestions based on their understanding and review goals. This interaction pattern, especially prevalent among expert users in our study, highlights the value of adaptable, goal-driven review interfaces that go beyond one-size-fits-all automation.

Overall, these findings suggest that while AI-assisted review tools have advanced the automation frontier, their practical effectiveness hinges on aligning with developer intentions, review contexts, and team conventions. MACEDON contributes to this ongoing effort by offering a dimensionally scaffolded review process that reflects how real developers reason about and communicate code quality.

6.8.2 Code Optimization in Programming Differs from Traditional Software Refactoring

Code optimization in programming tasks, especially in practical workflows, differs significantly from traditional software refactoring practices. While traditional refactoring typically emphasizes systematic transformation of code structure, including improving modularity, reducing duplication, or renaming for better readability [50], we found that optimization in real-world scenarios is often goal-driven, non-linear, and context-specific.

In our study, participants did not treat optimization as a purely structural clean-up task. Instead, they strategically selected which aspects to prioritize based on their goals. For instance, our experimental data and expert case study revealed a recurring workflow pattern where participants first optimized for clarity and redundancy before focusing on time and space efficiency. This sequential approach highlights a key distinction from refactoring guidelines that typically apply rule-based transformations uniformly [130]. One participant noted: "If the code isn't understandable, then optimizing for speed won't help. Clarity comes first." Such remarks reinforce that optimization is not merely mechanical improvement, but an iterative and interpretative process.

Moreover, optimization decisions were often informed by the intended audience or use case. When participants optimized code for instructional or peer-review scenarios, they focused heavily on clarity and documentation. In contrast, when optimizing for deployment or performance benchmarks, participants shifted attention to low-level algorithmic improvements. This context-aware behavior is rarely supported by traditional refactoring tools, which assume a fixed notion of "better" code, often guided by style guides or design patterns [6].

MACEDON was designed to support this flexible optimization process by providing dimension-specific feedback across five key aspects: clarity, redundancy, time efficiency,

space efficiency, and documentation. Rather than enforcing uniform refactoring templates, the system allowed users to select dimensions that aligned with their intent. Our evaluation showed that workflows beginning with structural dimensions (clarity/redundancy) produced code with higher accuracy and readability scores, echoing similar calls for context-sensitive tooling in modern IDEs [18].

These findings suggest that programming optimization workflows are not simply micro-refactorings; they are inherently interpretive, shaped by the programmer's goals, audience, and timing. This also has implications for tool design: systems should move beyond static rules and instead scaffold decision-making with contextual and composable suggestions, much like MACEDON does through its multi-dimensional design.

Furthermore, our observations speak to broader discussions in software engineering and HCI about adaptive support for complex workflows [20, 96]. As programming becomes increasingly collaborative and domain-specific, future tooling should reflect the diversity of optimization goals rather than converge on one-size-fits-all models. Supporting exploration, decision tracking, and even divergent optimization strategies may ultimately prove more useful than enforcing consistency alone.

Beyond these workflow-focused observations, we observed a mismatch between objective performance numbers and participants' feelings of improvement. Experts found many optimized versions produced with MACEDON only slightly faster or lighter in memory, yet participants still felt they had made meaningful progress. We think this mismatch is partly due to MACEDON's real-time scoring interface, which shows separate scores for each dimension after every edit. Even small score increases gave users quick feedback, reinforcing their sense of control. Such timely cues can amplify the perceived value of minor gains and should be integral to future optimization tool design. In summary, our study reveals that code optimization in real-world programming differs substantially from traditional software refactoring in both structure and intent. Tools like MACEDON, which account for the multiplicity of goals and adapt to the workflow at hand, are better suited to support modern programming practices.

6.8.3 Limitations and Future Work

Our study has several limitations. First, we focused on novice programmers and short C++ snippets, which may limit generalizability to professional developers or real-world software repositories. Future work should investigate MACEDON's effectiveness on professional-scale codebases and in team-based development settings.

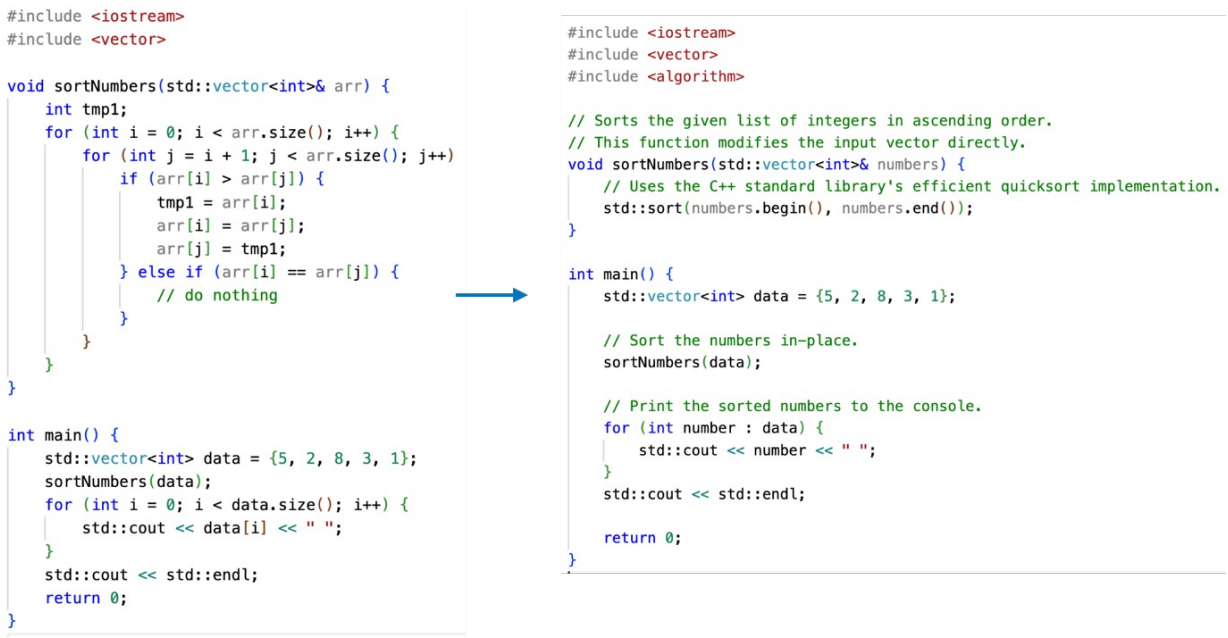
Second, while we evaluated five optimization dimensions, our system currently performs single-objective optimizations. Exploring multi-objective tradeoffs and dynamic prioritization remains an important next step. In real-world scenarios, programmers often need to balance competing goals such as improving performance without sacrificing clarity. Future work could incorporate interactive mechanisms that allow users to define or adjust their optimization priorities based on context or task-specific constraints.

Additionally, while our system surfaces explainable suggestions through structured UI, we did not explore deeper explainability techniques. Future research could investigate how to visualize reasoning behind recommendations or simulate "what-if" scenarios (e.g., "what happens if I accept this suggestion?") This would increase trust and educational value, particularly in classroom or mentoring settings

Finally, although our study revealed interaction-level behaviors, we did not assess long-term learning effects. Future longitudinal studies could examine whether MACEDON helps users internalize optimization principles or risks inducing over-reliance. These directions are critical for advancing MACEDON's educational and practical impact.

6.9 Conclusion

We presented MACEDON, a Visual Studio Code extension designed to support the *evaluation and optimization* stage by helping programmers with code optimization through real-time, multi-dimensional feedback. The system serves as both a practical tool and a research probe for understanding how developers make trade-offs across different dimensions of code quality including clarity, redundancy, time efficiency, space efficiency, and documentation. The design of MACEDON was guided by a formative study involving six programmers with varying levels of experience, along with a data-driven analysis on a C++ code benchmark. Our controlled user study and two case studies show that MACEDON significantly reduced task completion time while maintaining or enhancing the accuracy, readability, and satisfaction of the final code.



```cpp
#include <iostream>
#include <vector>

void sortNumbers(std::vector<int>& arr) {
    int tmp1;
    for (int i = 0; i < arr.size(); i++) {
        for (int j = i + 1; j < arr.size(); j++)
            if (arr[i] > arr[j]) {
                tmp1 = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp1;
            } else if (arr[i] == arr[j]) {
                // do nothing
            }
        }
    }
}

int main() {
    std::vector<int> data = {5, 2, 8, 3, 1};
    sortNumbers(data);
    for (int i = 0; i < data.size(); i++) {
        std::cout << data[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// Sorts the given list of integers in ascending order.
// This function modifies the input vector directly.
void sortNumbers(std::vector<int>& numbers) {
    // Uses the C++ standard library's efficient quicksort implementation.
    std::sort(numbers.begin(), numbers.end());
}

int main() {
    std::vector<int> data = {5, 2, 8, 3, 1};

    // Sort the numbers in-place.
    sortNumbers(data);

    // Print the sorted numbers to the console.
    for (int number : data) {
        std::cout << number << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Figure 6.4: Optimized version of a student's sorting algorithm with improvements guided by MACEDON. Detailed optimization by dimension: (1) **Clarity**: Variable names such as `arr` and `tmp1` were replaced with more descriptive names like `numbers` and `number`, improving readability. The loop to print sorted values was changed to a range-based for loop for more concise and clearer iteration. (2) **Time Efficiency**: The original nested loop sorting algorithm with time complexity $O(n^2)$ was replaced by the C++ standard library function `std::sort`, which uses an optimized quicksort algorithm, improving the sorting time to O(n log n). (3) **Space Efficiency**: Unnecessary temporary variables (e.g., `tmp1`) and redundant conditional checks (e.g., `arr[i] == arr[j]`) were removed. The space overhead was reduced as no extra memory was used for the sorting process. (4) **Documentation**: Added comments explaining the function's purpose, the choice of sorting algorithm, and the logic of the iteration and output steps. These comments make the code easier to understand for beginners and future developers. (5) **Redundancy**: A redundant conditional branch `else if (arr[i] == arr[j])` was removed because it had no functional effect, simplifying the logic and reducing unnecessary operations.

Chapter 7

Conclusion

7.1 Summary of Contributions

This dissertation contributes to the foundations of *interactive human–AI collaboration* for programming workflows [69], and to the design of systems that support documentation, presentation, and evaluation and optimization within existing tools and practices. The contributions are organized below into *artifact contributions* (the four interactive systems developed) and *empirical research contributions* (the findings from user studies, benchmarks, and analyses).

7.1.1 Artifact Contributions

This dissertation introduces four interactive systems that address distinct stages of the programming workflow through *human–AI collaboration*. *HAConvGNN* provides an interactive human–AI collaboration system (a Jupyter plugin) for understanding code through context-aware interpretations that authors can inspect, refine, and validate; to enable it, the project first constructs the notebookCDG dataset and develops the HAConvGNN model for context-aware documentation generation across multiple cells. *NBDESCRIB* provides an interactive human–AI collaboration system for collaborative writing; to enable it, the project first establishes a task formulation and benchmark for purpose-driven documentation generation from code and its execution results (using tabular output as a common example), with explicit guidelines for different communicative purposes. *Slide4N* creates interactive presentation slides directly from multi-code-cell notebooks while maintaining

author control over content selection and layout, exemplifying human–AI collaboration in the presentation stage. *MACEDON* extends the IDE with real-time, multi-dimensional evaluation of code quality (clarity, documentation, redundancy, runtime, and memory), making trade-offs visible during development and guiding programmers through collaborative optimization.

These systems achieve their goals through complementary technical approaches. HAConvGNN introduces a hierarchical attention mechanism and graph-augmented encoder that encodes each code cell as an AST graph, aggregates cross-cell signals, and attends to salient tokens for notebook-style markdown generation. NBDESCRIB establishes a benchmarking pipeline for purpose-driven documentation generation with 15 guideline categories and systematic error analyses, quantifying the challenge when code and its execution results (e.g., tabular output) are considered jointly. Slide4N operates on the invariant components of notebooks (code cells, outputs, and text) to distill content, map it to slide templates, and expose iterative refinement controls. MACEDON combines static analysis and lightweight runtime profiling to compute five quality dimensions continuously and surfaces them via inline annotations and summary dashboards.

7.1.2 Empirical Research Contributions

The implementation and validation of these *human–AI collaborative* systems yielded several empirical contributions.

A human evaluation found that participants rated HAConvGNN-generated documentation significantly higher than sequence and graph baselines on correctness, informativeness, and readability, and that a Jupyter plugin incorporating HAConvGNN reminded programmers to document code they would have ignored while reducing documentation development time through effective human–AI collaboration. An ablation study on a curated Kaggle corpus confirmed that both the graph encoding and cross-cell attention contribute to performance, with the largest gains when multiple cells are considered jointly. Benchmarks on NBDESCRIB revealed that state-of-the-art models, including large language models, achieve strong fluency but exhibit frequent factual errors around numbers and aggregation; fine-tuned CodeT5 can outperform larger models when guided by explicit purpose-driven guidelines. A two-part user study found that participants produced slides significantly more efficiently with Slide4N than with their usual manual workflow, valued the ability to override AI suggestions, and reported highest satisfaction when the system provided a strong starting point requiring only minor edits—a pattern consistent with effective human–AI collaboration. A within-subject experiment found that participants completed the optimization task faster and were more satisfied with the final code

when using MACEDON compared with a standard LLM-based code assistant, and that the multi-dimensional view changed editing priorities by making documentation and readability improvements visible alongside performance trade-offs, demonstrating how human–AI collaboration can enhance real-time development. Case studies highlighted that MACEDON's evaluations help programmers locate issues earlier, guide refactoring decisions with better context, and communicate code quality to collaborators.

7.2 Supporting Programming Workflows: Authors, Tools, or Human–AI Collaboration?

Programming workflows can be supported at different stages and by different actors. Understanding, documentation, presentation, and evaluation and optimization support may be produced by authors manually, by tool vendors through integrated features, or by human–AI collaborative systems. The central question is: where does support most effectively reside, and what are the trade-offs?

This dissertation focuses primarily on support that operates *at authoring time* within the programmer's existing environment—during understanding, documentation, presentation, and evaluation and optimization. But support could also be envisioned at earlier or later stages: during code review, during onboarding, or when artifacts are consumed by downstream stakeholders. A simplified lifecycle spans authors who produce artifacts manually, tool vendors who provide integrated features, and human–AI collaborative systems (this dissertation) that operate within existing tools while keeping authors in control.

Author-driven support. It is natural to assume that authors, who own the code and understand the intent, should understand code, produce documentation, create presentations, and evaluate and optimize quality themselves. Prior work has often focused on helping authors write better documentation or build more informative reports [63, 171, 187]. This dissertation argues, however, that the burden on authors is high: documentation becomes outdated as code evolves [167]; creating slides from notebooks is laborious and repetitive [221]; and juggling multiple evaluation dimensions (correctness, readability, performance) requires separate tools and mental overhead. Authors also cannot easily anticipate what level of detail or which representations will best serve future readers or collaborators.

Tool-vendor support. An alternative is to shift support to tool vendors—IDE and notebook platform providers who can integrate documentation hints, slide generation, or quality dashboards directly into their products. Author submission of source code and outputs could be processed by the platform to produce multiple representations, analogous to publishers who transform manuscript submissions into multiple formats. Some platforms already move in this direction: Jupyter offers extensions for execution and visualization [95]; VS Code integrates linting and profiling [18]. The advantage is reduced author effort; the limitation is that vendors must generalize across diverse workflows and may not support domain-specific or custom evaluation criteria without significant engineering investment.

Human–AI collaborative support. A third approach is human–AI collaboration that operates *within* existing tools without requiring platform-level changes. This is the approach taken in this dissertation: HAConvGNN, NBDESCRIB, Slide4N, and MACEDON all operate on invariant components—code cells, execution outputs, and text—that are exposed by standard notebook and IDE interfaces. AI proposes transformations (documentation, slide content, quality scores); authors accept, modify, or reject. The benefit is that support can be adopted incrementally and tailored to individual projects, while authors retain agency over what gets generated [69]. As demonstrated throughout this thesis, such information-equivalent representations can still meaningfully improve productivity, confidence, and the ability to share work [190].

7.3 Integration and Coherence

The four projects form a coherent progression that addresses the thesis statement: interactive human–AI collaboration can enhance key stages of programming workflows by supporting understanding code, documenting code, presenting code and results, and evaluating and optimizing quality, within existing tools and practices and with authors in control.

HAConvGNN (Chapter 3) established that modeling cross-cell relationships improves documentation quality in notebooks. Documentation often needs to describe both code and execution results; taking tabular output as a common example, NBDESCRIB (Chapter 4) addresses purpose-driven documentation generation and representation through an interactive human–AI collaboration system. Building on HAConvGNN's notebook-structure insights, Slide4N (Chapter 5) leverages this structure to extract and organize content for presentation slides. The progression from post-hoc analysis to real-time evaluation reflects

how addressing documentation and communication needs reveals the importance of evaluation during development, leading to MACEDON (Chapter 6).

All four projects operate on invariant components—code cells, execution outputs, and text—ensuring compatibility with common practices and integration with existing tools. Through human–AI collaboration, the systems generate context-aware documentation, help communicate computational results, and provide real-time multi-dimensional evaluation, without forcing authors to change their workflows.

7.4 User Requirements and Evolving AI Capabilities

7.4.1 User Populations Across Stages

The four projects recruit different user populations: data scientists and machine learning practitioners for HAConvGNN and NBDESCRIB, data scientists and graduate students for Slide4N, and novice C++ programmers for MACEDON. This variation is intentional. Each workflow stage serves a different purpose and attracts users with different backgrounds. Understanding and documentation tasks arise most naturally in exploratory data analysis, where notebook users with domain expertise need to explain code and outputs to collaborators. Presentation tasks span audiences from technical peers to non-technical stakeholders, requiring communication skills alongside technical knowledge. Evaluation and optimization tasks are common in software development courses and professional practice, where programmers at various experience levels seek to improve code quality.

These differences raise the question of what AI literacy or domain expertise users need to collaborate effectively at each stage. For the understanding stage, users must be able to judge whether AI-generated documentation correctly describes what code does; this requires reading comprehension and some familiarity with the programming language. For the documentation stage, users must identify factual errors in generated descriptions, especially when numbers and aggregation are involved; this demands attention to detail and knowledge of the data being described. For the presentation stage, users must assess whether slide content communicates the intended message to the target audience; this requires both technical understanding and communication judgment. For the evaluation and optimization stage, users must interpret quality scores and decide which trade-offs matter for their context; this requires at least basic knowledge of performance metrics. In all cases, the Propose–Inspect–Refine loop lowers the entry barrier: users do not need to produce artifacts from scratch, but they do need enough expertise to evaluate and correct AI output.

The required expertise varies by stage, which suggests that future systems could adapt their level of explanation or guidance based on user background.

7.4.2 Pipeline Robustness Under Advancing AI Capabilities

Recent advances in AI-powered development tools have produced systems that can handle multiple programming tasks end-to-end. Such systems can write code, generate documentation, and suggest optimizations within a single interaction. As these capabilities mature, the boundaries between the four stages addressed in this dissertation may appear to blur.

However, the stage-based decomposition remains valuable for three reasons. First, each stage has a different purpose and a different quality standard. Documentation records design intent for future reference; presentations communicate results to a broader group; evaluation feedback guides the programmer during active development. A single end-to-end system may not satisfy all these purposes equally, because the tone, level of detail, and verification criteria differ across stages. Second, the stage decomposition provides natural checkpoints for human review. Even when AI capabilities increase, programmers benefit from inspecting intermediate artifacts rather than accepting a fully automated result [69]. The Propose–Inspect–Refine loop introduced in this dissertation can operate at a coarser granularity, for example, reviewing a complete documentation draft rather than individual sentences, but the need for human oversight does not disappear. Third, the modular design allows each stage to be improved independently. Advances in code understanding models do not require changes to the presentation or evaluation components, and vice versa. This modularity also makes it easier to integrate new techniques as they become available.

When AI systems become capable of handling entire workflows, the human role may shift from operator to reviewer. Programmers would spend less time steering individual steps and more time evaluating whether the overall result meets their intent and audience needs. The Propose–Inspect–Refine pattern accommodates this shift: the "propose" step grows larger (an entire workflow output rather than a single artifact), but the "inspect" and "refine" steps remain essential. This evolution is consistent with broader trends in human–AI interaction, where increasing automation calls for stronger verification and oversight mechanisms rather than eliminating human involvement.

7.4.3 What Remains Human and What Becomes Automated

Current large language models can generate code summaries, write documentation drafts, produce slide outlines, and suggest code improvements. These capabilities overlap with the

tasks addressed in this dissertation. However, the division of labor between human and machine differs across stages, and important tasks remain beyond the reach of automation.

For *understanding*, AI systems can generate code summaries and highlight dependencies across files or cells. What remains human is judging whether the generated explanation matches the programmer's mental model of program behavior. A summary may be fluent and syntactically correct yet miss the programmer's actual question about why certain design choices were made. The programmer must also decide what level of detail is needed and which parts of the codebase deserve closer attention.

For *documentation*, AI systems can produce fluent prose from code and execution outputs. What remains human is verifying factual accuracy, especially for numerical values, aggregation results, and causal claims. Our NBDESCRIB benchmarks show that even strong models make frequent factual errors on numbers and aggregation (Chapter 4). The programmer must also decide which communicative purpose the documentation serves and adjust tone and detail for the intended reader.

For *presentation*, AI systems can extract key content from notebooks and arrange it into slide templates. What remains human is selecting what information matters for the target audience, structuring the narrative to support a specific argument, and ensuring the presentation does not overstate or misrepresent results. These are communication decisions that depend on context the AI system does not have access to.

For *evaluation and optimization*, AI systems can compute quality metrics across multiple dimensions and suggest code transformations. What remains human is deciding which trade-offs to accept. A programmer may choose to keep less efficient code because it is more readable, or to tolerate some redundancy because removing it would complicate maintenance. These decisions depend on project requirements, team conventions, and deployment constraints that AI systems cannot fully access.

In summary, as AI capabilities advance, the tasks that shift to machines are primarily generative: drafting text, computing metrics, and proposing transformations. The tasks that remain human are evaluative and contextual: judging correctness against intent, selecting information for a specific audience, and making trade-off decisions. The interactive systems in this dissertation are designed for this division of labor. They automate the generative tasks and provide structured interfaces for the evaluative tasks that require human judgment.

7.5 Broader and Longer-Term Impact

Beyond the immediate contributions to programming workflows, this dissertation offers three broader impacts that extend to future system design and human–AI collaboration research.

The Propose–Inspect–Refine loop as a general design pattern. The Propose–Inspect–Refine loop is not specific to programming. Any creative or analytical workflow where a human expert must produce artifacts under time pressure can benefit from AI-generated drafts that the expert then reviews and edits. Examples include scientific writing, legal document drafting, medical report generation, and engineering design review. In each case, the core structure is the same: the AI generates a starting point, the human evaluates it against domain knowledge and intent, and the human refines the output. This dissertation provides evidence that the pattern works across four different artifact types (documentation, descriptions, slides, and quality reports) and four different purposes (building understanding, recording rationale, communicating results, and guiding improvement). This breadth suggests that the pattern can transfer to other domains where similar conditions hold: the task is partially automatable, the output requires expert judgment, and the expert benefits from a structured starting point rather than a blank page.

Stage-based decomposition as a design methodology. This dissertation decomposes programming workflows into stages defined by their primary artifact and purpose. This decomposition serves as a design methodology: it identifies where to build support by asking what artifact is produced, what purpose it serves, and what quality criteria apply. Future system designers can apply this methodology to other complex workflows. For example, a data analysis pipeline could be decomposed into data cleaning, exploratory analysis, modeling, and reporting, with each stage supported by a dedicated human–AI tool. The key insight is that different stages have different verification requirements, and a single tool that tries to handle all stages may not satisfy any of them well. Stage-based decomposition makes these differences explicit and enables targeted support.

Evolving human roles in AI-assisted work. As AI systems become more capable, the human role in collaborative workflows will shift from producing artifacts to evaluating and steering AI-produced artifacts. This dissertation provides an early framework for this shift. The systems developed here already position the programmer as a reviewer who accepts, modifies, or rejects AI output. As AI output quality improves, the review task changes:

programmers spend less time correcting errors and more time making high-level decisions about scope, audience, and trade-offs. The Propose–Inspect–Refine loop accommodates this evolution because the granularity of the "propose" step can grow while the "inspect" and "refine" steps remain under human control. This suggests that the verification and control mechanisms established in this dissertation will remain relevant even as the underlying AI models improve. The long-term value of this work lies not in the specific models used but in the interaction patterns and design principles that keep humans in control of AI-assisted workflows.

7.6 Limitations and Future Work

While our results provide evidence that interactive human–AI collaboration can support programming workflows, there are several limitations that may require further investigation.

7.6.1 Validation Gaps Across the Four Projects

The four *human–AI collaboration* projects differ in how rigorously they were validated.

Only HAConvGNN was tested in a controlled setting with an A-versus-B protocol: human evaluators rated documentation quality on correctness, informativeness, and readability against sequence and graph baselines. The Jupyter plugin that incorporates HAConvGNN went through pilot testing; participants reported that it reminded them to document code they would have ignored and reduced documentation time. We did not validate these benefits experimentally within this thesis; a formal evaluation is reported separately [188].

For NBDESCRIB, benchmarks and human evaluation show that state-of-the-art models achieve strong fluency but often make factual errors, especially on numbers and aggregation. Purpose-driven guidelines improve performance, but we did not validate whether the generated descriptions actually help downstream readers understand the code and tables.

Slide4N was tested for usability and satisfaction; participants produced slides more efficiently and valued the ability to override AI suggestions. It is unclear whether the AI-generated slide content helps audience comprehension. We did not conduct a comparative A-versus-B study, because no appropriate, widely available baseline system exists and fully manual slide creation differs substantially in task scope. This is a common issue with

human–AI collaboration systems [108, 185]: many studies focus on efficiency and satisfaction, yet testing the actual impact on end users (readers of documentation, audiences of slides, or stakeholders reviewing code) remains difficult.

For MACEDON, a within-subject experiment with 24 novice programmers compared the system against a standard LLM-based code assistant; participants completed tasks faster and were more satisfied. Case studies with an undergraduate and a senior engineer illustrated real-world use. In some cases, however, experts rated the optimized code as only slightly faster or lighter in memory, yet participants felt they had made meaningful progress. This mismatch suggests that measures like satisfaction and perceived improvement may not fully capture objective quality gains [185].

Future work should address these validation gaps directly. Studies that recruit downstream readers of NBDESCRIB-generated descriptions could measure comprehension and task completion (e.g., answering questions about the code or table). For Slide4N, audience-focused experiments similar to those in presentation research could test whether viewers retain key messages from AI-generated slides versus manually created ones. Longitudinal studies over several weeks could assess whether MACEDON changes how programmers prioritize optimization dimensions or whether perceived gains persist when experts rate code blindly. Cell2Doc [131] and related notebook documentation work have begun to report reader-oriented metrics; extending this to our pipeline would strengthen the evidence base.

7.6.2 Proxy Metrics and Evaluation Complexity

Parts of this thesis, like many studies on human–AI collaboration, rely on proxy measures: task completion time, satisfaction ratings, and Likert-scale scores. These may not give the full picture of long-term productivity, code quality, or the effectiveness of collaborative workflows.

Automatic metrics for text generation, such as ROUGE and BERTScore [118, 224], have been shown to correlate poorly with human judgments on tasks that require faithfulness or diversity [126]. We use these metrics in NBDESCRIB and HAConvGNN, alongside human evaluation where possible, but the link between automatic scores and real-world usefulness is still unclear.

The four systems also combine many components (graph encoding, cross-cell attention, slide templates, multi-dimensional scoring) that work together through iterative design. It is hard to tell which aspects contribute most, and limited evidence exists on which features matter most for usability versus measurable outcomes [108].

Future work on evaluation methodology could pursue several paths. Recent table-to-text research shows that fixing input problems (e.g., non-standard table layouts, misaligned headers and values) reduces factual errors by 52–76% in ToTTo-style tasks [177]; applying similar input normalization to NBDESCRIB could improve both automatic scores and faithfulness. Content-matching losses and fact-verification modules have been proposed for faithful generation; integrating these into the NBDESCRIB pipeline could yield metrics that better predict reader comprehension. Ablation studies that systematically remove or substitute components (e.g., graph encoding vs. sequence-only, different slide templates) would clarify which design choices matter for each project.

Across all four projects, our human evaluations rely primarily on Likert-scale ratings and summary scores. While these measures are standard in HCI research, they have known limitations. A single composite score can mask variation across evaluation dimensions, and ordinal Likert data may not satisfy the assumptions of parametric tests such as ANOVA without careful justification. Future studies should complement Likert-scale ratings with behavioral measures (e.g., time spent editing AI output, number of revisions accepted versus rejected) and with qualitative analysis of think-aloud protocols. For studies involving multiple evaluation dimensions, reporting individual dimension scores alongside composites, and using non-parametric tests where sample sizes are small, would strengthen the statistical conclusions.

7.6.3 When AI Assistance May Be Unhelpful

In human–AI collaboration, if AI-generated content or suggestions are not useful, they may be harmful [185]. For example, programmers might spend time editing low-quality docs or slide content instead of writing their own, or rely on dashboards that distract from the main issues. How often this happens is difficult to measure and may depend as much on the user as on the system.

This thesis mitigates some of these risks by design: documentation, slides, and suggestions are generated on demand and stay under author control. The collaborative paradigm (authors accept, modify, or reject AI output rather than receiving it automatically) helps preserve agency even when suggestions fall short.

Future work could investigate when and how to surface or suppress AI suggestions. Trust calibration research for AI-assisted refactoring suggests that users need ways to validate LLM output before accepting changes; similar mechanisms could be applied to documentation and optimization suggestions. Logging when users dismiss or heavily edit AI proposals could identify failure modes without requiring new user studies. Adaptive thresholds that

reduce suggestion frequency when the user repeatedly rejects output, or that vary detail level by expertise, may reduce the risk of unhelpful assistance while preserving benefits for receptive cases.

7.6.4 Generalizability

The findings on human–AI collaboration may not generalize beyond the populations and contexts we studied.

Participants in the HAConvGNN human evaluation had data science and machine learning backgrounds ($N = 15$); NBDESCRIB involved graduate students with data science experience ($N = 10$); Slide4N recruited data scientists and graduate students ($N = 18$); and MACEDON recruited novice C++ programmers ($N = 24$) who use LLM-based code assistants. The datasets are largely from Kaggle notebooks and C++ code samples; the work focuses on computational notebooks and Python/C++. Results may not generalize to other document types, languages, IDEs, or workflow stages.

Future work could extend each project in more targeted ways. For HAConvGNN, recent work on integrating code metrics (cyclomatic complexity, coupling) as auxiliary signals has shown gains for notebook documentation; combining such metrics with our graph encoder could improve quality on notebooks with complex control flow. Extending HAConvGNN beyond Python to R or Julia notebooks would test whether the cross-cell dependency structure generalizes across ecosystems. For NBDESCRIB, automatically inferring guideline categories from existing markdown (e.g., via clustering or few-shot classification) would reduce the need for manual annotation when scaling to new domains. Improving numeric reasoning in generated descriptions (e.g., via chain-of-thought decoding or table-aware pretraining) could address the aggregation and value errors observed in our benchmarks. For Slide4N, outline-first workflows (where users specify slide structure before content retrieval) have been proposed to better support message organization; integrating such a step could address the gap between cell selection and narrative design. Support for audience-adaptive content (e.g., simpler summaries for non-technical viewers vs. technical detail for peers) would align with findings that audience background affects comprehension of AI-generated explanations. For MACEDON, extending beyond C++ to other languages (e.g., Python, JavaScript) would require language-specific static analysis and profiling hooks, but would test whether the multi-dimensional evaluation framework transfers. Benchmarks for IDE-based agents on real-world software engineering tasks could compare MACEDON-style tools against other AI coding assistants.

Integration across projects also presents opportunities. MACEDON's real-time evalu-

ation could trigger NBDESCRIB-style description updates when code or outputs change; NBDESCRIB's joint code-table descriptions could supply richer content for Slide4N slides. Broader research questions in human–AI collaboration remain open [69], including how programmers adapt their workflows when AI assistance is available and the long-term effects on code quality and maintainability.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. "Gpt-4 technical report." In: *arXiv preprint arXiv:2303.08774* (2023). DOI: 10.48550/arXiv.2303.08774.

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. "HPCToolkit: Tools for performance analysis of optimized parallel programs." In: *Concurrency and Computation: Practice and Experience*. Vol. 22. 6. Wiley Online Library, 2010, pp. 685–701.

[3] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. "Software documentation issues unveiled." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1199–1210.

[4] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. "Unified Pre-training for Program Understanding and Generation." In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, June 2021, pp. 2655–2668. DOI: 10.18653/v1/2021.naacl-main.211. URL: https://aclanthology.org/2021.naacl-main.211.

[5] Ibrahim Taha Aksu, Zhengyuan Liu, Min-Yen Kan, and Nancy Chen. "Velocidapter: Task-oriented dialogue comprehension modeling pairing synthetic text generation with domain adaptation." In: *Proceedings of the 22nd Annual Meeting of the Special Interest Group on Discourse and Dialogue*. 2021, pp. 133–143.

[6] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ó Cinnéide, Ali Ouni, and Yuanfang Cai. "An interactive and dynamic search-based approach to software refactoring recommendations." In: *IEEE Transactions on Software Engineering* 46.9 (2018), pp. 932–961.

[7] Uri Alon, Omer Levy, and Eran Yahav. "code2seq: Generating Sequences from Structured Representations of Code." In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=H1gKYo09tX.

[8] Chenxin An, Jiangtao Feng, Kai Lv, Lingpeng Kong, Xipeng Qiu, and Xuanjing Huang. "Cont: Contrastive neural text generation." In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 2197–2210.

[9] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. "Program synthesis with large language models." In: *arXiv preprint arXiv:2108.07732* (2021). DOI: 10.48550/arXiv.2108.07732.

[10] Damian Avila. *RISE*. 2019. URL: https://github.com/damianavila/RISE.

[11] Benjamin Bach, Zezhong Wang, Matteo Farinella, Dave Murray-Rust, and Nathalie Henry Riche. "Design patterns for data comics." In: *Proceedings of the 2018 chi conference on human factors in computing systems*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–12.

[12] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. "Codeplan: Repository-level coding using llms and planning." In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 675–698.

[13] Ori Bar El, Tova Milo, and Amit Somech. "Automatically generating data exploration sessions using deep reinforcement learning." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1527–1537.

[14] Shraddha Barke, Michael B James, and Nadia Polikarpova. "Grounded copilot: How programmers interact with code-generating models." In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (2023), pp. 85–111.

[15] Jeff Barnes. "Azure machine learning." In: *Microsoft Azure Essentials*. Washington, DC, USA: Microsoft, 2015.

[16] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. "The influence of non-technical factors on code review." In: *2013 20th working conference on reverse engineering (WCRE)*. IEEE. 2013, pp. 122–131.

[17] Kent Beck. "Embracing change with extreme programming." In: *Computer* 32.10 (1999), pp. 70–77.

[18] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. "Developer testing in the ide: Patterns, beliefs, and behavior." In: *IEEE Transactions on Software Engineering* 45.3 (2017), pp. 261–284.

[19] Matthias Blohm, Glorianna Jagfeld, Ekta Sood, Xiang Yu, and Ngoc Thang Vu. "Comparing attention-based convolutional and recurrent neural networks: Success and limitations in machine reading comprehension." In: *arXiv preprint arXiv:1808.08744* abs/1808.08744 (2018).

[20] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2009, pp. 1589–1598.

[21] Matthew Brehmer and Robert Kosara. "From jam session to recital: Synchronous communication and collaboration around data in organizations." In: *arXiv preprint arXiv:2107.09042* abs/2107.09042 (2021).

[22] John Brooke et al. "SUS-A quick and dirty usability scale." In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7.

[23] Ruven Brooks. "Towards a theory of the comprehension of computer programs." In: *International Journal of Man-Machine Studies* 18.6 (1983), pp. 543–554.

[24] G Ann Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.

[25] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. "MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation." In: *IEEE Transactions on Software Engineering* 49.7 (2023), pp. 3675–3691.

[26] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities." In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–12.

[27] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. "What's wrong with computational notebooks? Pain points, needs, and design opportunities." In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–12.

[28] David L Chen and Raymond J Mooney. "Learning to sportscast: a test of grounded language acquisition." In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 128–135.

[29] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. "Decision transformer: Reinforcement learning via sequence modeling." In: *Advances in neural information processing systems* 34 (2021), pp. 15084–15097.

[30] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. "Evaluating large language models trained on code." In: *arXiv preprint arXiv:2107.03374* (2021). DOI: 10.48550/arXiv.2107.03374.

[31] Wenhu Chen, Jianshu Chen, Yu Su, Zhiyu Chen, and William Yang Wang. "Logical natural language generation from open-domain tables." In: *arXiv preprint arXiv:2004.10404* (2020).

[32] Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. "Tabfact: A large-scale dataset for table-based fact verification." In: *arXiv preprint arXiv:1909.02164* (2019).

[33] Yu Chen, Lingfei Wu, and Mohammed J Zaki. "Reinforcement learning based graph-to-sequence model for natural question generation." In: *The Eighth International Conference on Learning Representations (ICLR 2020)*. 2020.

[34] Yung-Yu Chen, Yu Su, Hsin-Hsiang Chen, Yu-Ting Chung, Tzu-Fu Chen, Jian-Jia Chen, and Tei-Wei Kuo. "Speedoo: Prioritizing performance optimization opportunities." In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 60–70.

[35] Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. "HiTab: A hierarchical table dataset for question answering and natural language generation." In: *arXiv preprint arXiv:2108.06712* (2021).

[36] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Y. Bengio. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." In: (June 2014). DOI: 10.3115/v1/D14-1179.

[37] Harika Cm, Debasmita Das, Rajesh Kumar Ranjan, Siddhartha Asthana, et al. "LoKI: Money Laundering Report Generation via Logical Table-to-Text using Meta Learning." In: *Proceedings of the Fifth Workshop on Financial Technology and Natural Language Processing and the Second Multimodal AI For Financial Forecasting*. 2023, pp. 104–110.

[38] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. "Social coding in GitHub: Transparency and collaboration in an open software repository." In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. ACM, 2012, pp. 1277–1286. DOI: 10.1145/2145204.2145396.

[39] Hoa Trang Dang. "Overview of DUC 2005." In: *Proceedings of the document understanding conference*. Vol. 2005. Citeseer. 2005, pp. 1–12.

[40] Haijie Ding and Xiaolong Xu. "SAN-T2T: An automated table-to-text generator based on selective attention network." In: *Natural Language Engineering* (2023), pp. 1–25.

[41] David Donoho. "50 years of data science." In: *Journal of Computational and Graphical Statistics* 26.4 (2017), pp. 745–766.

[42] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. "Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists." In: *Proceedings of the 2020 CHI conference on human factors in computing systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–12.

[43] Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Heng Ping, Chenyu Zhou, Nesreen K Ahmed, Guixiang Ma, Mihai Capota, Theodore L Willke, Shahin Nazarian, et al. "Leveraging Reinforcement Learning and Large Language Models for Code Optimization." In: *arXiv preprint arXiv:2312.05657* (2023). DOI: 10.48550/arXiv.2312.05657.

[44] Roy Eisenstadt and Michael Elhadad. "Evaluation Guidelines to Deal with Implicit Phenomena to Assess Factuality in Data-to-Text Generation." In: *Proceedings of the 1st Workshop on Understanding Implicit and Underspecified Language*. Online: Association for Computational Linguistics, Aug. 2021, pp. 20–27. DOI: 10.18653/v1/2021.unimplicit-1.3. URL: https://aclanthology.org/2021.unimplicit-1.3.

[45] Will Epperson, Doris Jung-Lin Lee, Leijie Wang, Kunal Agarwal, Aditya G Parameswaran, Dominik Moritz, and Adam Perer. "Leveraging Analysis History for Improved In Situ Visualization Recommendation." In: *Computer Graphics Forum*. Vol. 41. USA: Wiley Online Library, 2022, pp. 145–155.

[46] Ilker Etikan, Sulaiman Abubakar Musa, Rukayya Sunusi Alkassim, et al. "Comparison of convenience sampling and purposive sampling." In: *American journal of theoretical and applied statistics* 5.1 (2016), pp. 1–4.

[47] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139. URL: https://aclanthology.org/2020.findings-emnlp.139.

[48] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. "My ai wants to know if this will be on the exam: Testing openai's codex on cs2 programming exercises." In: *Proceedings of the 25th Australasian Computing Education Conference*. 2023, pp. 97–104.

[49] Andrew Forward and Timothy C. Lethbridge. "The relevance of software documentation, tools and technologies: A survey." In: *Proceedings of the 2002 ACM Symposium on Document Engineering*. ACM, 2002, pp. 26–33.

[50] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[51] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. "InCoder: A Generative Model for Code Infilling and Synthesis." In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=vf-T4YuzIo.

[52] Nat Friedman. *Introducing GitHub Copilot: Your AI Pair Programmer. 2021*. 2021.

[53] Tsu-Jui Fu, William Yang Wang, Daniel McDuff, and Yale Song. "Doc2PPT: Automatic presentation slides generation from scientific documents." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. California, CA, USA: Association for the Advancement of Artificial Intelligence, 2022, pp. 634–642.

[54] Sainyam Galhotra, Udayan Khurana, Oktie Hassanzadeh, Kavitha Srinivas, Horst Samulowitz, and Miao Qi. "Automated Feature Enhancement for Predictive Modeling using External Knowledge." In: *2019 International Conference on Data Mining Workshops (ICDMW)*. New York, NY, USA: IEEE, 2019, pp. 1094–1097. DOI: 10.1109/ICDMW.2019.00161.

[55] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. "Search-Based LLMs for Code Optimization." In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025, pp. 578–590. DOI: 10.1109/ICSE55347.2025.00021.

[56] Yanjun Gao, Chen Sun, and Rebecca J. Passonneau. "Automated Pyramid Summarization Evaluation." In: *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 404–418. DOI: 10.18653/v1/K19-1038. URL: https://aclanthology.org/K19-1038.

[57] Heng Gong, Xiaocheng Feng, Bing Qin, and Ting Liu. "Table-to-Text Generation with Effective Hierarchical Encoder on Three Dimensions (Row, Column and Time)." In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2019, pp. 3141–3150.

[58] Google. *Cloud AutoML Custom Machine Learning Models*. 2021. URL: https://cloud.google.com/automl.

[59] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. "Graphcodebert: Pre-training code representations with data flow." In: *arXiv preprint arXiv:2009.08366* (2020).

[60] Philip J. Guo and Margo I. Seltzer. "BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure." In: *4th Workshop on the Theory and Practice of Provenance, TaPP'12, Boston, MA, USA, June 14-15, 2012*. Boston, MA, USA: USENIX Association, 2012.

[61] Zhixin Guo, Jianping Zhou, Jiexing Qi, Mingxuan Yan, Ziwei He, Guanjie Zheng, Zhouhan Lin, Xinbing Wang, and Chenghu Zhou. "Towards Controlled Table-to-Text Generation with Scientific Reasoning." In: *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2024, pp. 9951–9955.

[62] Vivek Gupta, Maitrey Mehta, Pegah Nokhiz, and Vivek Srikumar. "INFOTABS: Inference on Tables as Semi-structured Data." In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020, pp. 2309–2324. DOI: 10.18653/v1/2020.acl-main.210. URL: https://aclanthology.org/2020.acl-main.210.

[63] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. "Supporting program comprehension with source code summarization." In: *32nd ACM/IEEE International Conference on Software Engineering* (Jan. 2010). DOI: 10.1145/1810295.1810335.

[64] Jiawei Han, Jian Pei, and Hanghang Tong. *Data mining: concepts and techniques*. San Francisco, CA: Morgan kaufmann, 2022.

[65] Sakib Haque, Aakash Bansal, Lingfei Wu, and Collin McMillan. "Action word prediction for neural source code summarization." In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2021, pp. 330–341.

[66] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. "Improved automatic summarization of subroutines via attention to file context." In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 300–310.

[67] Andrew Head, Fred Hohman, Titus Barik, Steven Mark Drucker, and Robert DeLine. "Managing Messes in Computational Notebooks." In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*. Glasgow, Scotland, UK: ACM, 2019, p. 270.

[68] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. "Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs." In: *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. New York, USA: ACM, 2020, pp. 1–12.

[69] Jeffrey Heer. "Agency plus automation: Designing artificial intelligence into interactive systems." In: *Proceedings of the National Academy of Sciences* 116.6 (2019), pp. 1844–1850.

[70] Weixiang Hong, Kaixiang Ji, Jiajia Liu, Jian Wang, Jingdong Chen, and Wei Chu. "GilBERT: Generative Vision-Language Pre-Training for Image-Text Retrieval." In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '21. Virtual Event, Canada: Association

for Computing Machinery, 2021, 1379–1388. ISBN: 9781450380379. DOI: 10.1145/3404835.3462838.

[71] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. "Summarizing Source Code with Transferred API Knowledge." In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. IJCAI'18. Stockholm, Sweden: AAAI Press, 2018, 2269–2275. ISBN: 9780999241127.

[72] Yue Hu and Xiaojun Wan. "PPSGen: Learning-based presentation slides generation for academic papers." In: *IEEE transactions on knowledge and data engineering* 27.4 (2014), pp. 1085–1097.

[73] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search." In: *CoRR* abs/1909.09436 (2019). arXiv: 1909.09436. URL: http://arxiv.org/abs/1909.09436.

[74] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. "CodeSearchNet challenge: Evaluating the state of semantic code search." In: *arXiv preprint arXiv:1909.09436* (2019).

[75] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. "Summarizing Source Code using a Neural Attention Model." In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. DOI: 10.18653/v1/P16-1195. URL: https://www.aclweb.org/anthology/P16-1195.

[76] JetBrains. *IntelliJ IDEA: The Java IDE for Professional Developers*. Accessed: 2024-10-10. 2001. URL: https://www.jetbrains.com/.

[77] Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc Le, Yun-Hsuan Sung, Zhen Li, and Tom Duerig. "Scaling Up Visual and Vision-Language Representation Learning With Noisy Text Supervision." In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. Breckenridge, Colorado, USA: PMLR, 2021, pp. 4904–4916.

[78] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. "Discovering the syntax and strategies of natural language programming with generative language models." In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 2022, pp. 1–19.

[79] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones." In: *29th International Conference on Software Engineering (ICSE 2007)*. IEEE, 2007, pp. 96–105. DOI: 10.1109/ICSE.2007.30.

[80] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. "TreeBERT: A tree-based pre-trained model for programming language." In: *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. Ed. by Cassio de Campos and Marloes H. Maathuis. Vol. 161. Proceedings of Machine Learning Research. PMLR, 2021, pp. 54–63. URL: https://proceedings.mlr.press/v161/jiang21a.html.

[81] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=VTF8yNQM66.

[82] Project Jupyter. *Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science*. the Helmsley Trust, the Gordon, Betty Moore Foundation, and the Alfred P. Sloan Foundation. 2015. URL: https://blog.jupyter.org/project-jupyter-computational-narratives-as-the-engine-ofcollaborative-data-science-2b5fb94c3c58.

[83] Project Jupyter. *JupyterLab: the next generation of the Jupyter Notebook*. Project Jupyter. 2018. URL: https://blog.jupyter.org/jupyterlab-thenext-generation-of-the-jupyter-notebook-5c949dabea3.

[84] Project Jupyter. *nbconvert*. Project Jupyter. 2021. URL: https://github.com/jupyter/nbconvert.

[85] Mihir Kale and Abhinav Rastogi. "Template guided text generation for task-oriented dialogue." In: *arXiv preprint arXiv:2004.15006* (2020).

[86] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code." In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670. DOI: 10.1109/TSE.2002.1019480.

[87] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. "Pre-trained Contextual Embedding of Source Code." In: *ArXiv* abs/2001.00059 (2019).

[88] DaYe Kang, Tony Ho, Nicolai Marquardt, Bilge Mutlu, and Andrea Bianchi. "ToonNote: Improving communication in computational notebooks using interactive data comics." In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. New York, United States: Association for Computing Machinery, 2021, pp. 1–14.

[89] Mary Beth Kery and Brad A. Myers. "Exploring exploratory programming." In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Oct. 2017, pp. 25–29. DOI: 10.1109/VLHCC.2017.8103446.

[90] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. "The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. New York, United States: ACM, 2018, pp. 1–11.

[91] Junaed Younus Khan and Gias Uddin. "Automatic code documentation generation using gpt-3." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–6.

[92] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. "Automatic detection of performance regressions." In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 103–113.

[93] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In: *International Conference on Learning Representations* (Dec. 2014).

[94] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks." In: *arXiv preprint arXiv:1609.02907* (2016).

[95] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. *Jupyter Notebooks-a publishing format for reproducible computational workflows.* Vol. 2016. Virginia, USA: IOS Press, 2016.

[96] Amy J Ko, Robert DeLine, and Gina Venolia. "Information needs in collocated software development teams." In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 344–353.

[97] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks." In: *IEEE Transactions on software engineering* 32.12 (2006), pp. 971–987.

[98] Philipp Koehn and Rebecca Knowles. "Six Challenges for Neural Machine Translation." In: *Proceedings of the First Workshop on Neural Machine Translation*. Vancouver: Association for Computational Linguistics, Aug. 2017, pp. 28–39. DOI: 10.18653/v1/W17-3204. URL: https://aclanthology.org/W17-3204.

[99] Matthias Kraus, Nicolas Wagner, and Wolfgang Minker. "Modelling and predicting trust for developing proactive dialogue strategies in mixed-initiative interaction." In: *Proceedings of the 2021 International Conference on Multimodal Interaction*. 2021, pp. 131–140.

[100] Sean Kross and Philip J Guo. "Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges." In: *Proceedings of the 2019 CHI conference on human factors in computing systems*. New York, USA: ACM, 2019, pp. 1–14.

[101] Sean Kross and Philip Guo. "Orienting, Framing, Bridging, Magic, and Counseling: How Data Scientists Navigate the Outer Loop of Client Collaborations in Industry and Academia." In: *Proceedings of the ACM on Human-Computer Interaction* 5.CSCW2 (2021), pp. 1–28.

[102] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[103] Thomas D LaToza, Gina Venolia, and Robert DeLine. "Maintaining mental models: a study of developer work habits." In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 492–501.

[104] A. LeClair and C. McMillan. "Recommendataions for Datasets for Source Code Summarization." In: *2019 Annual Conference of the North Americal Chapter of the Association for Computational Linguistics (NAACL)*. 2019.

[105] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. "Improved Code Summarization via a Graph Neural Network." In: *2020 IEEE International Conference on Program Comprehension (ICPC)*. 2020.

[106] Erin LeDell and Sebastien Poirier. "H2O AutoML: Scalable Automatic Machine Learning." In: *Proceedings of the AutoML Workshop at ICML*. Vol. 2020. Freiburg and Hannover, Germany: automl.org, 2020.

[107] Rémi Lebret, David Grangier, and Michael Auli. "Neural Text Generation from Structured Data with Application to the Biography Domain." In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 1203–1213. DOI: 10.18653/v1/D16-1128. URL: https://aclanthology.org/D16-1128.

[108] Chris van der Lee, Albert Gatt, Emiel Miltenburg, and Emiel Krahmer. "Human evaluation of automatically generated text: Current trends and best practice guidelines." In: *Computer Speech Language* 67 (May 2021), p. 101151. DOI: 10.1016/j.csl.2020.101151.

[109] Stanley Letovsky. "Cognitive processes in program comprehension." In: *Journal of Systems and software* 7.4 (1987), pp. 325–339.

[110] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension." In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020, pp. 7871–7880. DOI: 10.18653/v1/2020.acl-main.703. URL: https://aclanthology.org/2020.acl-main.703.

[111] Tongliang Li, Lei Fang, Jian-Guang Lou, and Zhoujun Li. "TWT: Table with written text for controlled data-to-text generation." In: *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2021, pp. 1244–1254.

[112] Xingjun Li, Yuanxin Wang, Hong Wang, Yang Wang, and Jian Zhao. "NBSearch: Semantic Search and Visual Exploration of Computational Notebooks." In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Vol. abs/2102.01275. New York, USA: ACM, 2021, pp. 1–14.

[113] Xingjun Li, Yizhi Zhang, Justin Leung, Chengnian Sun, and Jian Zhao. *EDAssistant: Supporting Exploratory Data Analysis in Computational Notebooks with In-Situ Code Search and Recommendation*. 2021. DOI: 10.48550/ARXIV.2112.07858.

[114] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. "Competition-level code generation with alphacode." In: *Science* 378.6624 (2022), pp. 1092–1097.

[115] Jenny T Liang, Chenyang Yang, and Brad A Myers. "A large-scale survey on the usability of ai programming assistants: Successes and challenges." In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024, pp. 1–13.

[116] Percy Liang, Michael Jordan, and Dan Klein. "Learning Semantic Correspondences with Less Supervision." In: *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Lan-*

guage Processing of the AFNLP. Suntec, Singapore: Association for Computational Linguistics, Aug. 2009, pp. 91–99. URL: https://aclanthology.org/P09-1011.

[117] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries." In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://www.aclweb.org/anthology/W04-1013.

[118] Chin-Yew Lin. "Rouge: A package for automatic evaluation of summaries." In: *Text summarization branches out*. 2004, pp. 74–81.

[119] Sherry Lin, Winthrop F Gillis, Caleb Weinreb, Ayman Zeine, Samuel C Jones, Emma M Robinson, Jeffrey Markowitz, and Sandeep Robert Datta. "Characterizing the structure of mouse behavior using motion sequencing." In: *arXiv preprint arXiv:2211.08497* (2022).

[120] Denghui Liu, Chi Xu, Wenjun He, Zhimeng Xu, Wenqi Fu, Lei Zhang, Jie Yang, Zhihao Wang, Bing Liu, Guangdun Peng, et al. "AutoGenome: an autoML tool for genomic research." In: *Artificial Intelligence in the Life Sciences* 1 (2021), p. 100017.

[121] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation." In: *Advances in Neural Information Processing Systems* 36 (2024).

[122] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. ""What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models." In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–31.

[123] Tianyu Liu, Kexiang Wang, Lei Sha, Baobao Chang, and Zhifang Sui. "Table-to-text generation by structure-aware seq2seq learning." In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[124] Xuye Liu, Dakuo Wang, April Yi Wang, Yufang Hou, and Lingfei Wu. "HAConvGNN: Hierarchical Attention Based Convolutional Graph Neural Network for Code Documentation Generation in Jupyter Notebooks." In: *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*. Pennsylvania, USA: Association for Computational Linguistics, 2021, pp. 4473–4485.

[125] Xuye Liu, Dakuo Wang, April Wang, Yufang Hou, and Lingfei Wu. "HAConvGNN: Hierarchical Attention Based Convolutional Graph Neural Network for Code Documentation Generation in Jupyter Notebooks." In: *Findings of the Association for Computational Linguistics: EMNLP 2021*. Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 4473–4485. DOI: 10.18653/v1/2021.findings-emnlp.381. URL: https://aclanthology.org/2021.findings-emnlp.381.

[126] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. *G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment*. 2023. arXiv: 2303.16634 [cs.CL].

[127] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. "Studying the usage of text-to-text transfer transformer to support code-related tasks." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. New York, NY, USA: IEEE, 2021, pp. 336–347.

[128] Richard E Mayer. *Multimedia learning*. Cambridge university press, 2001.

[129] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. "An empirical study of the impact of modern code review practices on software quality." In: *Empirical Software Engineering* 21 (2016), pp. 2146–2189.

[130] Tom Mens and Tom Tourwé. "A survey of software refactoring." In: *IEEE Transactions on software engineering* 30.2 (2004), pp. 126–139.

[131] Tamal Mondal, Scott Barnett, Akash Lal, and Jyothi Vedurada. "Cell2Doc: ML Pipeline for Generating Documentation in Computational Notebooks." In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 384–396.

[132] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. "Reading between the lines: Modeling user behavior and costs in AI-assisted programming." In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 2024, pp. 1–16.

[133] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. "ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification." In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 2332–2354.

[134] Michael J Muller, April Yi Wang, Steven I Ross, Justin D Weisz, Mayank Agarwal, Kartik Talamadupula, Stephanie Houde, Fernando Martinez, John T Richards, Jaimie Drozdal, et al. "How Data Scientists Improve Generated Code Documentation in Jupyter Notebooks." In: *IUI Workshops*. 2021.

[135] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q Vera Liao, Casey Dugan, and Thomas Erickson. "How data science workers work with data: Discovery, capture, curation, design, creation." In: *Proceedings of the 2019 CHI conference on human factors in computing systems*. New York, USA: ACM, 2019, pp. 1–15.

[136] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. "Refactoring: How do software engineers use it?" In: *2006 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE. 2006, pp. 511–514.

[137] Andrew Myers. *In Human-Centered AI, the Boundaries Between UX and Software Roles Are Evolving*. Stanford University. 2022. URL: https://hai.stanford.edu/news/human-centered-ai-boundaries-between-ux-and-software-roles-are-evolving.

[138] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. "Programmers are users too: Human-centered methods for improving programming tools." In: *Computer* 49.7 (2016), pp. 44–52.

[139] Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryściński, Hailey Schoelkopf, Riley Kong, Xiangru Tang, et al. "Fetaqa: Free-form table question answering." In: *Transactions of the Association for Computational Linguistics* 10 (2022), pp. 35–49.

[140] Nhan Nguyen and Sarah Nadi. "An empirical evaluation of GitHub Copilot's code suggestions." In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 1–5.

[141] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis." In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=iaYcJKpY2B_.

[142] Jekaterina Novikova, Oliver Lemon, and Verena Rieser. "Crowd-sourcing NLG Data: Pictures Elicit Better Data." In: *Proceedings of the 9th International Natural Language Generation conference*. Edinburgh, UK: Association for Computa-

tional Linguistics, 2016, pp. 265–273. DOI: 10.18653/v1/W16-6644. URL: https://aclanthology.org/W16-6644.

[143] Microsoft Office. *Make Your PowerPoint Presentations Accessible to People with Disabilities*. 2021. URL: https://support.microsoft.com/en-us/office/make-your-powerpoint-presentations-accessible-to-people-with-disabilities-6f7772b2-2f33-4bd2-8ca7-dae3b2b3ef25.

[144] William F Opdyke. "Refactoring object-oriented frameworks." In: *University of Illinois at Urbana-Champaign* (1992).

[145] Allan Paivio. *Mental representations: A dual coding approach*. Oxford University Press, 1986.

[146] Ankur P Parikh, Xuezhi Wang, Sebastian Gehrmann, Manaal Faruqui, Bhuwan Dhingra, Diyi Yang, and Dipanjan Das. "ToTTo: A controlled table-to-text generation dataset." In: *arXiv preprint arXiv:2004.14373* (2020).

[147] Ankur P Parikh, Xuezhi Wang, Sebastian Gehrmann, Manaal Faruqui, Bhuwan Dhingra, Diyi Yang, and Dipanjan Das. "ToTTo: A controlled table-to-text generation dataset." In: *arXiv preprint arXiv:2004.14373* (2020).

[148] Bumjin Park and Jaesik Choi. "Identifying the Source of Generation for Large Language Models." In: *arXiv preprint arXiv:2407.12846* (2024).

[149] Panupong Pasupat and Percy Liang. "Compositional Semantic Parsing on Semi-Structured Tables." In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 1470–1480. DOI: 10.3115/v1/P15-1142. URL: https://aclanthology.org/P15-1142.

[150] Nancy Pennington. "Stimulus structures and mental representations in expert comprehension of computer programs." In: *Cognitive Psychology* 19.3 (1987), pp. 295–341.

[151] Fernando Pérez and Brian E Granger. "IPython: A system for interactive scientific computing." In: *Computing in Science & Engineering* 9.3 (2007), pp. 21–29. DOI: 10.1109/MCSE.2007.53.

[152] Jeffrey M Perkel. "Why Jupyter is data scientists' computational notebook of choice." In: *Nature* 563.7732 (2018), pp. 145–147.

[153] David Piorkowski, Soya Park, April Yi Wang, Dakuo Wang, Michael Muller, and Felix Portnoy. "How AI Developers Overcome Communication Challenges in a Multidisciplinary Team: A Case Study." In: *Proceedings of the ACM on Human-Computer Interaction* 5.CSCW1 (2021), pp. 1–25.

[154] Russell A Poldrack, Thomas Lu, and Gašper Beguš. "AI-assisted coding: Experiments with GPT-4." In: *arXiv preprint arXiv:2304.13187* (2023). DOI: 10.48550/arXiv.2304.13187.

[155] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter F Tichy. "Quantitative assessments of development techniques: A case study." In: *IEEE Software* 17.4 (2000), pp. 44–51.

[156] Ruchir Puri et al. "CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks." In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. 2021. URL: https://openreview.net/forum?id=6vZVBkCDrHT.

[157] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2019. DOI: 10.48550/ARXIV.1910.10683.

[158] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2020. arXiv: 1910.10683 [cs.LG].

[159] Maithra Raghu, Katy Blumer, Greg Corrado, Jon Kleinberg, Ziad Obermeyer, and Sendhil Mullainathan. "The algorithmic automation problem: Prediction, triage, and human effort." In: *arXiv preprint arXiv:1903.12220* (2019). DOI: 10.48550/arXiv.1903.12220.

[160] Karthikeyan Ramasamy, Michael Muller, and Dakuo Wang. "Visualising the Invisible: Understanding Data Scientists' Documentation Practices in Jupyter Notebooks." In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2023, pp. 1–15.

[161] Kyle Richardson, Sina Zarrieß, and Jonas Kuhn. "The code2text challenge: Text generation in source code libraries." In: 2017.

[162] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. "The programmer's assistant: Conversational interaction with a large language model for software development." In: *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 2023, pp. 491–514.

[163] Winston W Royce. "Managing the development of large software systems: concepts and techniques." In: *Proceedings of the 9th international conference on Software Engineering*. 1987, pp. 328–338.

[164] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. "Code llama: Open foundation models for code." In: *arXiv preprint arXiv:2308.12950* (2023).

[165] Adam Rule, Ian Drosos, Aurélien Tabard, and James D Hollan. "Aiding collaborative reuse of computational notebooks with annotated cell folding." In: *Proceedings of the ACM on Human-Computer Interaction* 2.CSCW (2018), pp. 1–12.

[166] Adam Rule, Aurélien Tabard, and James D. Hollan. "Exploration and Explanation in Computational Notebooks." In: CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018, 1–12. ISBN: 9781450356206. DOI: 10.1145/3173574.3173606. URL: https://doi.org/10.1145/3173574.3173606.

[167] Adam Rule, Aurélien Tabard, and James D Hollan. "Exploration and explanation in computational notebooks." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–12.

[168] Simone Scalabrino, Gabriele Bavota, Barbara Russo, Rocco Oliveto Penta, and Andrea Marcus. "A comprehensive model for code readability." In: *Journal of Software: Evolution and Process* 30.12 (2018), e1958.

[169] Alexander G Shypula et al. "Learning Performance-Improving Code Edits." In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=ix7rLVHXyY.

[170] Jonathan Sillito, Gail C Murphy, and Kris De Volder. "Questions programmers ask during software evolution tasks." In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2006, pp. 23–34.

[171] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. "A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques." In: *IEEE Access* PP (July 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2931579.

[172] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. "Calibration and Correctness of Language Models for Code." In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025, pp. 540–552. DOI: 10.1109/ICSE55347.2025.00040.

[173] Margaret-Anne Storey. "Theories, tools and research methods in program comprehension: Past, present and future." In: *Software Quality Journal* 14.3 (2006), pp. 187–208.

[174] Lya Hulliyyatus Suadaa, Hidetaka Kamigaito, Kotaro Funakoshi, Manabu Okumura, and Hiroya Takamura. "Towards Table-to-Text Generation with Numerical Reasoning." In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Online: Association for Computational Linguistics, Aug. 2021, pp. 1451–1465. DOI: 10.18653/v1/2021.acl-long.115. URL: https://aclanthology.org/2021.acl-long.115.

[175] Krishna Subramanian, Johannes Maas, and Jan Borchers. "Tractus: Understanding and supporting source code experimentation in hypothesis-driven data science." In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. New York, USA: ACM, 2020, pp. 1–12.

[176] Edward Sun, Yufang Hou, Dakuo Wang, Yunfeng Zhang, and Nancy Xin Ru Wang. "D2S: Document-to-Slide Generation Via Query-Based Text Summarization." In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*. Pennsylvania, USA: Association for Computational Linguistics, 2021, pp. 1405–1418.

[177] Barkavi Sundararajan, Yaji Sripada, and Ehud Reiter. "Improving Factual Accuracy of Neural Table-to-Text Output by Addressing Input Problems in ToTTo." In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Mexico City, Mexico: Association for Computational Linguistics, 2024, pp. 7350–7376.

[178] Harini Suresh, Steven R Gomez, Kevin K Nam, and Arvind Satyanarayan. "Beyond expertise and roles: A framework to characterize the stakeholders of interpretable machine learning and their needs." In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. New York, USA: ACM, 2021, pp. 1–16.

[179] John Sweller. "Cognitive load during problem solving: Effects on learning." In: *Cognitive science* 12.2 (1988), pp. 257–285.

[180] Gareth Terry, Nikki Hayfield, Victoria Clarke, Virginia Braun, et al. "Thematic analysis." In: *The SAGE handbook of qualitative research in psychology* 2.17-37 (2017), p. 25.

[181] Adam Tornhill and Markus Borg. "Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases." In: *Proceedings of the International Conference on Technical Debt*. 2022, pp. 11–20.

[182] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. "Llama: Open and efficient foundation language models." In: *arXiv preprint arXiv:2302.13971* (2023).

[183] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and removal of feature envy bad smells." In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, pp. 519–523.

[184] Maria Tsiakmaki, Georgios Kostopoulos, Sotiris Kotsiantis, and Omiros Ragos. "Implementing AutoML in educational data mining for prediction tasks." In: *Applied Sciences* 10.1 (2019), p. 90.

[185] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models." In: *Chi conference on human factors in computing systems extended abstracts*. 2022, pp. 1–7.

[186] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. "Neural program repair by jointly learning to localize and repair." In: *7th International Conference on Learning Representations, ICLR 2019*. 2019.

[187] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Xuye Liu, Soya Park, Steve Oney, and Christopher Brooks. "What Makes a Well-Documented Notebook? A Case Study of Data Scientists' Documentation Practices in Kaggle." In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–7.

[188] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. "Themisto: Towards Automated Documentation Generation in Computational Notebooks." In: *arXiv preprint arXiv:2102.12592* (2021).

[189] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. "Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks." In: *ACM Trans. Comput.-Hum. Interact.* 29.2 (Jan. 2022). ISSN: 1073-0516. DOI: 10.1145/3489465. URL: https://doi.org/10.1145/3489465.

[190] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. "Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks." In: *ACM Transactions on Computer-Human Interaction* 29.2 (2022), pp. 1–33.

[191] April Yi Wang, Dakuo Wang, Xuye Liu, Lingfei Wu, et al. "Graph-Augmented Code Summarization in Computational Notebooks." In: *IJCAI*. 2021, pp. 5020–5023.

[192] April Yi Wang, Dakuo Wang, Xuye Liu, Lingfei Wu, et al. "Graph-Augmented Code Summarization in Computational Notebooks." In: *IJCAI'21 Demo* (2021).

[193] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. "Callisto: Capturing the" Why" by Connecting Conversations with Computational Narratives." In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. New York, USA: ACM, 2020, pp. 1–13.

[194] Dakuo Wang, Q Vera Liao, Yunfeng Zhang, Udayan Khurana, Horst Samulowitz, Soya Park, Michael Muller, and Lisa Amini. "How Much Automation Does a Data Scientist Want?" In: *arXiv preprint arXiv:2101.03970* (2021).

[195] Dakuo Wang, Lingfei Wu, Xuye Liu, Yi Wang, Chuang Gan, Jing Xu, Xue Ying Zhang, and Jun Wang. *Learning-based automated machine learning code annotation with graph neural network*. US Patent App. 17/088,018. 2022.

[196] Dakuo Wang, Lingfei Wu, Xuye Liu, Yi Wang, Chuang Gan, Jing Xu, Xue Ying Zhang, Jun Wang, and Jing James Xu. *Learning-based automated machine learning code annotation with graph neural network*. US Patent 11,928,156. 2024.

[197] Dakuo Wang, Lingfei Wu, Yi Wang, Xuye Liu, Chuang Gan, Si Er Han, Bei Chen, and Ji Hui Yang. *Learning-based automation machine learning code annotation in computational notebooks*. US Patent App. 17/069,402. 2022.

[198] Fengjie Wang, Xuye Liu, Oujing Liu, Ali Neshati, Tengfei Ma, Min Zhu, and Jian Zhao. "Slide4N: Creating Presentation Slides from Computational Notebooks with Human-AI Collaboration." In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–18.

[199] Hao Wang, Xiaodong Zhang, Shuming Ma, Xu Sun, Houfeng Wang, and Mengxiang Wang. "A Neural Question Answering Model Based on Semi-Structured Tables." In: *Proceedings of the 27th International Conference on Computational Linguistics*. Santa Fe, New Mexico, USA: Association for Computational Linguistics, Aug. 2018, pp. 1941–1951. URL: https://aclanthology.org/C18-1165.

[200] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation." In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*. 2021.

[201] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation." In: *arXiv preprint arXiv:2109.00859* abs/2109.00859 (2021).

[202] Yun Wang, Zhida Sun, Haidong Zhang, Weiwei Cui, Ke Xu, Xiaojuan Ma, and Dongmei Zhang. "DataShot: Automatic generation of fact sheets from tabular data." In: *IEEE transactions on visualization and computer graphics* 26.1 (2019), pp. 895–905.

[203] Yun Wang, Zhida Sun, Haidong Zhang, Weiwei Cui, Ke Xu, Xiaojuan Ma, and Dongmei Zhang. "DataShot: Automatic Generation of Fact Sheets from Tabular Data." In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 895–905. DOI: 10.1109/TVCG.2019.2934398.

[204] Zezhong Wang, Shunming Wang, Matteo Farinella, Dave Murray-Rust, Nathalie Henry Riche, and Benjamin Bach. "Comparing effectiveness and engagement of data comics and infographics." In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. New York, USA: ACM, 2019, pp. 1–12.

[205] Zijie J Wang, Katie Dai, and W Keith Edwards. "StickyLand: Breaking the Linear Presentation of Computational Notebooks." In: *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. New York, USA: ACM, 2022, pp. 1–7.

[206] Justin D Weisz, Mohit Jain, Narendra Nath Joshi, James Johnson, and Ingrid Lange. "BigBlueBot: teaching strategies for successful human-agent interactions." In: *Proceedings of the 24th International Conference on Intelligent User Interfaces*. 2019, pp. 448–459.

[207] John Wenskovitch, Jian Zhao, Scott Carter, Matthew Cooper, and Chris North. "Albireo: An interactive tool for visually summarizing computational notebook structure." In: *2019 IEEE visualization in data science (VDS)*. IEEE. New York, NY, USA: IEEE, 2019, pp. 1–10.

[208] Sam Wiseman, Stuart Shieber, and Alexander Rush. "Challenges in Data-to-Document Generation." In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 2253–2263. DOI: 10.18653/v1/D17-1239. URL: https://aclanthology.org/D17-1239.

[209] Robert F Woolson. "Wilcoxon signed-rank test." In: *Wiley encyclopedia of clinical trials* (2007), pp. 1–3.

[210] Xueqing Wu, Jiacheng Zhang, and Hang Li. "Text-to-Table: A New Way of Information Extraction." In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Pennsylvania, USA: Association for Computational Linguistics, 2022, pp. 2518–2533.

[211] Yifan Wu, Joseph M Hellerstein, and Arvind Satyanarayan. "B2: Bridging code and interactive visualization in computational notebooks." In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. New York, USA: ACM, 2020. DOI: 10.1145/3379337.3415851.

[212] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. "Measuring program comprehension: A large-scale field study with professionals." In: *IEEE Transactions on Software Engineering* 44.10 (2017), pp. 951–976.

[213] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. "IDE-based code generation: A case study of GitHub Copilot." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 1–12.

[214] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. "Graph2seq: Graph to sequence learning with attention-based neural networks." In: *arXiv preprint arXiv:1804.00823* (2018).

[215] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. "Graph2seq: Graph to sequence learning with attention-based neural networks." In: *arXiv preprint arXiv:1804.00823* (2018).

[216] Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. "Large language models are versatile decomposers: Decompose evidence and questions for table-based reasoning." In: *arXiv preprint arXiv:2301.13808* (2023).

[217] Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. "CoLadder: Manipulating Code Generation via Multi-Level Blocks." In: *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. UIST '24. Pittsburgh, PA, USA: Association for Computing Machinery, 2024. ISBN: 9798400706288. DOI: 10.1145/3654777.3676357. URL: https://doi.org/10.1145/3654777.3676357.

[218] Tao Yu et al. "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task." In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Ed. by Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii. Brussels, Belgium: Association for Computational Linguistics, 2018, pp. 3911–3921. DOI: 10.18653/v1/D18-1425. URL: https://aclanthology.org/D18-1425/.

[219] Nicholas Zakas. *ESLint: Pluggable JavaScript linter*. Accessed: 2024-10-10. 2013. URL: https://eslint.org/.

[220] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. "Large Language Models Meet NL2Code: A Survey." In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 7443–7464. DOI: 10.18653/v1/2023.acl-long.411. URL: https://aclanthology.org/2023.acl-long.411/.

[221] Amy X Zhang, Michael Muller, and Dakuo Wang. "How do Data Science Workers Collaborate? Roles, Workflows, and Tools." In: *arXiv preprint arXiv:2001.06684* (2020).

[222] Haowei Zhang, Shengyun Si, Yilun Zhao, Lujing Xie, Zhijian Xu, Lyuhao Chen, Linyong Nan, Pengcheng Wang, Xiangru Tang, and Arman Cohan. "OpenT2T: An Open-Source Toolkit for Table-to-Text Generation." In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2024, pp. 259–269.

[223] Tianjun Zhang, Fangchen Liu, Justin Wong, Pieter Abbeel, and Joseph E. Gonzalez. "The Wisdom of Hindsight Makes Language Models Better Instruction Followers." In: *Proceedings of the 40th International Conference on Machine Learning*. Ed. by Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 41414–41428. URL: https://proceedings.mlr.press/v202/zhang23ab.html.

[224] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. "Bertscore: Evaluating text generation with bert." In: *arXiv preprint arXiv:1904.09675* (2019).

[225] Yichi Zhang. "Detecting Code Comment Inconsistencies using LLM and Program Analysis." In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 2024, pp. 683–685.

[226] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. "Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code." In: *Transactions on Machine Learning Research* (2024). ISSN: 2835-8856. URL: https://openreview.net/forum?id=hkNnGqZnpa.

[227] Jian Zhao, Shenyu Xu, Senthil Chandrasegaran, Chris Bryan, Fan Du, Aditi Mishra, Xin Qian, Yiran Li, and Kwan-Liu Ma. "ChartStory: Automated partitioning, layout, and captioning of charts into comic-style narratives." In: *arXiv preprint arXiv:2103.03996* 29.2 (2021), pp. 1384–1399.

[228] Yilun Zhao, Zhenting Qi, Linyong Nan, Lorenzo Jaime Yu Flores, and Dragomir Radev. "LoFT: enhancing faithfulness and diversity for table-to-text generation via logic form control." In: *arXiv preprint arXiv:2302.02962* (2023).

[229] Yilun Zhao, Zhenting Qi, Linyong Nan, Boyu Mi, Yixin Liu, Weijin Zou, Simeng Han, Ruizhe Chen, Xiangru Tang, Yumo Xu, et al. "QTSumm: Query-Focused Summarization over Tabular Data." In: *The 2023 Conference on Empirical Methods in Natural Language Processing*. 2023.

[230] Chengbo Zheng, Dakuo Wang, April Yi Wang, and Xiaojuan Ma. "Telling Stories from Computational Notebooks: AI-Assisted Presentation Slides Creation for Presenting Data Science Work." In: *CHI Conference on Human Factors in Computing Systems*. New York, USA: ACM, 2022, pp. 1–20.

[231] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x." In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2023, pp. 5673–5684.

[232] Ingrid Zukerman and Diane Litman. "Natural language processing and user modeling: Synergies and limitations." In: *User modeling and user-adapted interaction* 11.1 (2001), pp. 129–158.

APPENDICES

Appendix A

HACONVGNN CODE-DOCUMENTATION EXAMPLES

A.1 Code Snippets-Documentation Pair Examples

Documentation	
Ground Truth	Feature scaling
Our Model	Feature scaling
Code2seq	We can have the model
Graph2seq	The next step is a lot of the training set
T5-small	Scaling
Code Cells	

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(train_df)
train_scale = pd.DataFrame(scaler.transform(train_df))
```

Table A.1: Example: Feature Scaling

Documentation	
Ground Truth	handle missing values in X test
Our Model	we can deal with missing values
Code2seq	We can have the categorical data
Graph2seq	We can also make any numeric variable in the model
T5-small	Filling the missing values in the test set
Code Cells	
`cols_with_missing_val = [col for col in X_test.columns if X_test[col].isnull().any()]` `print(cols_with_missing_val)`	

`from sklearn.impute import SimpleImputer`
`my_imputer = SimpleImputer(strategy='most_frequent')`
`my_imputer.fit(X_train)`
`imputed_X_test = pd.DataFrame(my_imputer.transform(X_test))`
`imputed_X_test.columns = X_test.columns`	

Table A.2: Example: Handle Missing Values

Documentation	
Ground Truth	Data Augmentation
Our Model	Data Builder
Code2seq	We can have the model
Graph2seq	LSTM
T5-small	Visualize the images

Code Cells

```
import warnings
from imgaug import augmenters as iaa
warnings.filterwarnings("ignore")

augmentation = iaa.Sequential([
        iaa.OneOf([ ## rotate
            iaa.Affine(rotate=0),
            iaa.Affine(rotate=90),
            iaa.Affine(rotate=180),
            iaa.Affine(rotate=270),
        ]),

        iaa.Fliplr(0.5),
        iaa.Flipud(0.2),

        iaa.OneOf([
            iaa.Cutout(fill_mode="constant", cval=255),
            iaa.CoarseDropout((0.0, 0.05), size_percent=(0.02, 0.25)),
            ]),

        iaa.OneOf([
            iaa.Snowflakes(flake_size=(0.2, 0.4), speed=(0.01, 0.07)),
            iaa.Rain(speed=(0.3, 0.5)),
        ]),

        iaa.OneOf([
            iaa.Multiply((0.8, 1.0)),
            iaa.contrast.LinearContrast((0.9, 1.1)),
        ]),

        iaa.OneOf([
            iaa.GaussianBlur(sigma=(0.0, 0.1)),
            iaa.Sharpen(alpha=(0.0, 0.1)),
        ])
    ],
    random_order=True
)
```

```
def get_ax(rows=1, cols=1, size=7):
    _, ax = plt.subplots(rows, cols, figsize=(size*cols, size*rows))
    return ax

limit = 4
ax = get_ax(rows=2, cols=limit//2)

for i in range(limit):
    image, image_meta, class_ids,\
    bbox, mask = modellib.load_image_gt(
        dataset_train, config, image_id, use_mini_mask=False,
        augment=False, augmentation=augmentation)

    visualize.display_instances(image, bbox, mask, class_ids, dataset_train.class_names, ax=ax[i//2, i
    ↪  % 2, show_mask=False, show_bbox=False)
```


Table A.2: Example Data Augmentation

Documentation	
Ground Truth	Plot the model's performance
Our Model	Plot the model's performance
Code2seq	We can have the model
Graph2seq	The next step is a lot of the training and test set
T5-small	Plot model performance

Code Cells

```
plt.plot(history_size_val_1)
plt.plot(history_size_val_2)
plt.plot(history_size_val_3)
plt.plot(history_size_val_4)
plt.plot(history_size_val_5)
plt.plot(history_size_val_6)
plt.title('Model accuracy for different Conv sizes')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.ylim(0.98,1)
plt.xlim(0,n_epochs)
plt.legend(['8-16', '16-32', '32-32', '24-48', '32-64', '48-96', '64,128'], loc='upper left')
plt.savefig('convolution_size.png')
plt.show()
```

Table A.4: Example: Plot Model Performance

Documentation	
Ground Truth	Count Monthly Mean
Our Model	Monthly Count
Code2seq	We can have a look at the training set
Graph2seq	Feature Engineering
T5-small	Creating a new column

Code Cells

```
for year in year_list:
    for month in range(num_months_per_year):
        start_date = datetime.datetime(year, month+1, 1, 0, 0, 0)
        end_date = datetime.datetime(year, month+1, 19, 23, 0, 0)
        count_mean = train_data[start_date:end_date]
        ['count'].mean()
        train_data.loc[start_date:end_date, 'count_mean'] = count_mean

        start_date = datetime.datetime(year, month+1, 20, 0, 0, 0)
        last_day_of_month = calendar.monthrange(year,month+1)[1]
        end_date = datetime.datetime(year, month+1, last_day_of_month, 23, 0, 0)
        test_data.loc[start_date:end_date, 'count_mean'] = count_mean
```

```
test_data.head()
```

Table A.5: Example: Count Monthly Mean

Appendix B

NBDESCRIB HUMAN EVALUATION DETAILS

B.1 Human Evaluation

We also conduct a human evaluation to further evaluate whether those models can generate reasonable and oriented text with our dataset.

Participants: Our human evaluation task involves reading the code snippet, its output table, and a guideline description and rating the generated documentation from them. We recruited 10 participants (6 male, 4 female) who are fluent English speakers with around six years of experience in the data science and machine learning field. We conducted a rigorous qualification process, evaluating their knowledge of coding practices and data analysis, to ensure high-quality annotations. We hired them by sending invited emails to graduate students who have experience in data science work. We allocated up to 90 minutes for each participant to complete the study, and for their valuable time and input, each participant received a compensation of $20.

Task: We randomly selected 50 pairs of documentation and code from MACEDON. Note that each pair has only one code, one table, and one guideline description, but may have one descriptive text. Each participant is assigned 50 pairs. Each pair is evaluated by 10 individuals. In each trial, a participant reads 8 candidate descriptions for the same code snippet-table-guideline: one by GPT-4o with chain-of-thought, one by GPT-4o with in-context-learning, one as the ground truth, and another five by five different models. The order of these eight is also randomized, so participants do not know which descriptive text

Model	Correctness	Orientation	Readability
Groundtruth	$\overline{x}$ = 1.19, σ=1.32	$\overline{x}$ = 1.45, σ=1.02	$\overline{x}$ = 1.61, σ=0.78
CodeT5-Large	$\overline{x}$ = -0.43, σ=1.55	$\overline{x}$ = 1.27, σ=1.11	$\overline{x}$ = 0.55, σ=1.60
GPT-4o	$\overline{x}$ = -0.42, σ=1.67	$\overline{x}$ = 1.18, σ=1.44	$\overline{x}$ = 0.53, σ=1.51
GPT-3	$\overline{x}$ = -0.41, σ=1.58	$\overline{x}$ = 0.98, σ=1.39	$\overline{x}$ = 0.51, σ=1.61
Llama3	$\overline{x}$ = -0.44, σ=1.41	$\overline{x}$ = 1.03, σ=1.42	$\overline{x}$ = 0.51, σ=1.82
CodeLlama	$\overline{x}$ = -0.61, σ=1.39	$\overline{x}$ = 0.88, σ=1.36	$\overline{x}$ = 0.52, σ=1.69
GPT-4o with chain-of-thought	$\overline{x}$ = -0.39, σ=1.54	$\overline{x}$ = 0.94, σ=1.35	$\overline{x}$ = 0.48, σ=1.66
GPT-4o with in-context-learning	$\overline{x}$ = -0.35, σ=1.60	$\overline{x}$ = 0.91, σ=1.28	$\overline{x}$ = 0.46, σ=1.83

Table B.1: Human Evaluation Result.

is from which model. The participant is asked to rate the 8 documentation texts along three dimensions using a five-point Likert scale from -2 to 2. For the human study, we adopted a -2 to 2 scale because it provides participants with a clear central point (0) to indicate neutrality, while the negative and positive ranges (-2 and 2) allow for stronger differentiation in agreement and disagreement. This scale helps human evaluators express nuanced opinions about correctness, orientation, and readability, reducing ambiguity and improving clarity during evaluation.

- *Correctness*: The generated description matches the code and table content.
- *Orientation*: The generated description is written in the correct guideline category.
- *Readability*: The generated description is in readable English grammar and words.

Evaluation Results: We conducted *Wilcoxon tests* [209] with a significance level of 0.05 to compare the performance of Ground Truth against CodeT5-Large, GPT-3, and GPT-4o in the Correctness, Orientation, and Readability dimensions. The Wilcoxon test is a non-parametric statistical test used to compare two paired groups of data. The obtained p-values indicate the probability of observing the reported differences if there were no true differences between the models. The results indicate significant differences in the Correctness dimension, where Ground Truth outperforms CodeT5-Large (V = 5628, p = 1.74e-30), GPT-3 (V = 5635, p = 5.46e-31), GPT-4o (V = 5647, p = 4.79e-30), Llama3 (V = 5732, p = 3.32e-30), and CodeLlama (V = 5948, p = 1.27e-30). It is also worth noting that CodeT5 performs slightly better than GPT-4o in terms of correctness from Table B.1, possibly because it handles code-containing data sets better.

Similarly, in the Orientation dimension, Ground Truth surpasses CodeT5-Large (V =

3567, p = 1.59e-20), GPT-3 (V = 3731, p = 1.77e-20), GPT-4o (V = 3559, p = 1.82e-20), Llama3 (V = 3722, p = 1.89e-20), and CodeLlama (V = 3883, p = 1.33e-20).

For the Readability dimension which considers whether the generated description is a valid English sentence, Ground Truth outperforms all models once again: CodeT5-Large (V = 4363, p = 1.40e-7), GPT-3 (V = 4030, p = 3.81e-14), GPT-4o (V = 4451, p = 3.01e-10), Llama3 (V = 4556, p = 2.38e-13), and CodeLlama (V = 4573, p = 1.62-15). It is also worth noting that GPT-4o with chain-of-thought and in-context-learning have worse performance than GPT-4o which demonstrates that guidelines can better assist the description generation for code and table.

The statistically significant p-values (all below 0.05) in each dimension demonstrate it is difficult to meet the correctness, orientation, and readability requirements of the user due to the difficulty of the task. Future work can be accomplished by designing an innovative model to address this challenge.

B.2 Guideline-Code Snippets-Table-Documentation Pair Examples

Guideline description

Aggregation: Calculate the descriptive statistical indicators (e.g., average, sum, count, etc.) based on the data attributes

Code Cells

```
for dataset in [titanic_train,titanic_test]:
    dataset['IsAlone'] = 0
    dataset.loc[dataset['FamilySize'] == 1, 'IsAlone'] = 1
titanic_train.head(3)
```

Table

	PassengerId	Survived	Sex	Ticket	Cabin
0	1	0	male	A/5 21171	NaN
1	2	1	female	PC 17599	C85
2	3	1	female	STON/O2.3101282	NaN

Documentation

ground truth	Cabin feature has 2 missing values
CodeT5-Large	These five passengers are in the same cabin
GPT-4o	It adds an IsAlone column to the Titanic datasets and displays the first three rows data
GPT-3	The cabin column has many null values
Llama3	Creates a new column to indicate if passengers are traveling alone or with family.
CodeLlama	Average fare paid by the passengers

Table B.2: An example of code and its table output pair(cabin is the Cabin number). In this case, the curated table only keeps the cabin columns.

Guideline description

Trend: Indicates a general tendency over a period of time

Code Cells

```
train[['Pclass', 'Survived']].groupby(['Pclass'],
↪  as_index=False).mean().sort_values(by='Survived', ascending=False)
```

Table

	Pclass	Survived
0	1	0.629630
1	2	0.472826
2	3	0.242363

Documentation

ground truth	Higher class survived more
CodeT5-Large	The survived rate for this new feature varies whether it is a tend to be a higher class passengers or a very high class passengers
GPT-4o	it presents the survival rates for each passenger class, demonstrating a clear pattern where lower class numbers correspond to higher survival rates
GPT-3	the survived rate for the pclass 1 is higher than the pclass 2
Llama3	First-class passengers having the highest survival rate and third-class passengers having the lowest.
CodeLlama	The most common class is 3rd class with 24% of the survived passengers

Table B.3: An example of code and its table output pair (Pclass is the ticket level. The smaller the value, the higher the class). In this case, the table is not curated because ground truth have class and survived variables.

Guideline description

Goal: Express user's goal. To say what value or function they tend to use for the later research

Code Cells

```
for dataset in [titanic_train,titanic_test]:
    dataset['FamilySize'] = dataset['SibSp'] + dataset['Parch'] + 1
titanic_train.head(3)
```

Table

	PassengerId	Survived	SibSp	Parch	IsAlone	FamilySize
0	1	0	1	0	0	2
1	2	1	1	0	0	2
2	3	1	0	0	1	2

Documentation

ground truth	Checking if the person is alone or with a family by checking the SibSp and Parch column in Titanic passenger data and add a FamilySize column in titanic_train and titanic_test datasets
CodeT5-Large	We can create another feature called IsAlone
GPT-4o	The goal is to create a new attribute called family size in both the titanic train and titanic test datasets
GPT-3	we can see that sib sp and parch are highly correlated
Llama3	Understand the relationships between different passenger characteristics in the Titanic dataset
CodeLlama	To predict the family size of a passenger in the Titanic disaster

Table B.4: An example of code and its table output pair. In this case, the curated table only keeps the passenger, SibSp, Parch, and FamilySize column.

Guideline description

Extreme: Identify the data cases that are the most extreme in relation to the data attributes or within a specific range

Code Cells

```
Tuned_rf = tune_model(rf)
```

Table

Model	Accuracy	AUC	Recall	...
0	0.7895	0.8864	0.6250	...
1	0.9474	1.000	0.8750	...
2	0.8947	0.9318	0.8750	...
3	0.7368	0.8523	1.0000	...
4	0.8947	0.8667	0.8889	...
5	0.9473	0.9444	0.8889	...
6	0.8947	0.9111	0.7778	...
7	0.7895	0.8333	0.6667	...
Mean	0.8617	0.9189	0.8222	...
SD	0.0675	0.0556	0.1348	...

Documentation

ground truth	Model 1 has the highest accuracy while the code tune with random forest
CodeT5-Large	The highest accuracy is 0.8442
GPT-4o	It shows the accuracy values for different sequence numbers where the mean accuracy is 0 8617 and the standard deviation is 0 0675
GPT-3	the highest accuracy is 0.7895
Llama3	Identify the most extreme data points in relation to their

Guideline description

Extreme: Identify the data cases that are the most extreme in relation to the data attributes or within a specific range

Code Cells

```
df = pd.read_csv("../input/red-wine-quality-cortez-et-al-2009/
winequality-red.csv")
df.head()
```

Table

Wine	fixed acidity	volatile acidity	pH	sulphates	alcohol	quality
0	7.4	0.70	4o1	0.56	9.4	5
1	7.8	0.88	3.20	0.68	9.8	5
2	7.8	0.76	3.26	0.65	9.8	5
3	11.2	0.28	3.16	0.58	9.8	5
4	7.4	0.70	4o1	0.56	9.4	5

Documentation

ground truth	The first red wine has the highest pH value
CodeT5-Large	the biggest ph is 3.20
GPT-4o	It determine how these values deviate significantly from the normal range, providing insights into outliers.
GPT-3	data frame sort ph values
Llama3	Identifies the top 4 most extreme data points for each attribute.
CodeLlama	The wine quality is a continuous variable with a range of 3 to 9

Table B.6: An example of code and its table output pair. In this case, the curated table only keeps the Wine and pH column.

B.3 Kaggle Competition Link

We crawled highly voted notebooks from seven top popular Kaggle competitions - House Price Prediction[1], Titanic Survival Prediction[2], Predict Future Sales[3], Spaceship Titanic[4], U.S. Patent Phrase to Phrase Matching[5], JPX Tokyo Stock Exchange Prediction[6], Ubiquant Market Prediction[7]

B.4 Detail of Baseline Models

CodeT5 is a large pre-trained encoder-decoder Transformer model that better leverages the code semantics conveyed from the developer-assigned identifiers. Since CodeT5 is a competitive code-related text generation model, when using this model in our task, we converted the relevant table and guideline category description into an inline comment in code and then fine-tuned the model. It has 770 million parameters and the computational budget is around 3 hours.

GPT-3 (Generative Pre-training Transformer 3) is an autoregressive language model with 175 billion parameters, 10x more than any previous non-sparse language model. To use the GPT3 model for our task, we combine guideline description, code, and table as input text. It has 175 billion parameters. The computational budget is around 1 hour. To use the GPT-3 model, we register an account on OpenAI and use the related API (openai api fine_tunes.create[8]) to fine-tune the GPT-3 model. Also, we built a dataset suitable for GPT-3 training, which can shared with the public.

GPT-4o is an advanced iteration of the GPT-3 model with around 12 billion parameters and a default backend of free ChatGPT. The computational budget is around 1 hour and 15 minutes. Its ability to comprehend context, generate coherent and contextually relevant responses, and perform a wide array of language-related tasks is further refined. It is an easily accessible tool and has been widely used in real life. So we add it as an advanced baseline.

[1] https://www.kaggle.com/c/house-prices-advanced-regression-techniques
[2] https://www.kaggle.com/c/titanic/
[3] https://www.kaggle.com/competitions/competitive-data-science-predict-future-sales
[4] https://www.kaggle.com/competitions/spaceship-titanic
[5] https://www.kaggle.com/competitions/us-patent-phrase-to-phrase-matching
[6] https://www.kaggle.com/competitions/jpx-tokyo-stock-exchange-prediction
[7] https://www.kaggle.com/competitions/ubiquant-market-prediction
[8] https://beta.openai.com/docs/guides/fine-tuning

Llama3 [182]. We use Llama3 3.1-70B in this task. It is an advanced language model with approximately 70 billion parameters. Its default backend is designed for efficiency and scalability. The computational budget for Llama3 is approximately 1 hour and 30 minutes. Its ability to understand context, generate coherent and contextually relevant responses, and perform a wide range of language-related tasks is significantly enhanced. Llama3 is a powerful and accessible tool, widely used in various applications. Therefore, it is included as an advanced baseline.

CodeLlama3 [164] is an advanced version of the Llama3 model. We use the 13B version in this task. The model aims to handle complex code generation and comprehension tasks efficiently. The computational budget of CodeLlama3 is about 2 hours. CodeLlama3 performs well in understanding intricate programming contexts, generating accurate and contextually appropriate code, and performing various code-related tasks. Its accessibility and versatility have made it a valuable tool for developers and researchers, serving as an advanced baseline for code-centric applications.

B.5 Guideline Categories

B.6 Prompt for Chain of Thought on GPT-4o

```
Given the 15 guidelines describing the code cell and its table output in
the Jupyter Notebook:
1. Value(Get the exact data attribute values for a set of criteria)
2. Difference(A comparison between at least two distinct attributes within
 the target object, or a comparison between the target object and
previously measured values)
3. Trend(Indicates a general tendency over a period of time)
4. Proportion(Measure the proportion of selected data attribute(s) within
a specified set )
5. Categorization(Select the data attribute(s) that meet the condition)
6. Distribution(Show the amount of shared value for the selected data
attributes or present a breakdown of all data attributes)
7. Rank(Sort data attributes by their values and display a breakdown of
selected attributes)
8. Association (Identify the useful relationship between two or more data
attributes)
9. Extreme(Identify the data cases that are the most extreme in relation
to the data attributes or within a specific range )
10. Outlier(Determine whether there are unexpected data attributes or
statistically significant outliers)
11. Aggregation(Calculate the descriptive statistical indicators (e.g.,
average, sum, count, etc. ) based on the data attributes.)
12. Goal(Express user's goal. To say what value or function they tend to
use for the later research)
13. Reason(Express reason using the data from the table or explain the
reasons why certain functions are used or why a task is performed.)
14. Feature Engineer(The process of selecting, transforming, extracting,
combining, and manipulating raw data to generate the desired variables for
 analysis or predictive modeling)
15. Complementary
Details (Express additional contextual elements
and supporting informa- tion to enhance
understanding of the primary content)
```

```
Q: When using Jupyter Notebook, the data scientist wants to write a
description in the Markdown cell covering the code cell and its table
output. The description should be less than 50 tokens.
Table Sequence:
     | passengerid| survived
mean| 446.000000 | 0.383838
Code: train = pd.read_csv("../input/titanic/train.csv")
    # take a quick look at the training data
    train.describe(include="all")"

A: The data scientist wants to write a description in Extreme guideline,
the description he writes is: the mean survived rate is 38.3 denoting most
 of the passengers have not survived

Q: When using Jupyter Notebook, the data scientist wants to write a
description in the Markdown cell covering the code cell and its table
output:
<Table>
<Code>
```

B.7 Prompt for In-Context Learning on GPT-4o

```
Q: When using Jupyter Notebook, the data scientist wants to write a
description in the Markdown cell covering the code cell and its table
output.
Table Sequence:
          | passengerid| survived
      mean| 446.000000| 0.383838
train = pd.read_csv("../input/titanic/train.csv")
# take a quick look at the training data
train.describe(include="all")"

A: The data scientist wants to write a description in Extreme guideline,
the description he writes is: the mean survived rate is 38.3 denoting most
 of the passengers have not survived
```

```
Q: When using Jupyter Notebook, the data scientist wants to write a
description in the Markdown cell covering the code cell and its table
output:
<Table>
<Code>
```

B.8 G-Eval in Coherence, Consistency, Correctness, and Fluency

B.8.1 Definition

G-EVAL is a prompt-based evaluation system with three main components: 1) a prompt defining the evaluation task and criteria, 2) a chain-of-thoughts (CoT) that includes intermediate instructions generated by the LLM to outline the detailed evaluation steps, and 3) a scoring function that uses the LLM to calculate scores based on the probabilities of the returned tokens. The prompt should also contain customized evaluation criteria for different NLG tasks.

Coherence The overall quality of all sentences working together. This aligns with the DUC [39] quality question on structure and coherence, which states that "the summary should be well-structured and well-organized, building from sentence to sentence to form a coherent body of information about a topic." Followed by [126], we conduct the G-EVAL in the four dimensions below:

Consistency The generated description is written in the correct guideline category. A factually consistent summary contains only statements that are supported by the source document. Annotators were instructed to penalize summaries containing hallucinated facts.

Fluency The quality of individual sentences. According to the DUC [39] quality guidelines, sentences in the summary "should have no formatting problems, capitalization errors, or obvious grammatical errors (e.g., fragments, missing components) that make the text difficult to read."

Correctness Compare the actual output directly with the expected output to verify factual accuracy. Check if all elements specified in the expected output are present and accurately represented in the actual output. Evaluate whether there are any discrepancies in details, values, or information between the two outputs.

B.8.2 Prompt Details for obtaining metrics

```
You will be given one description written for a code, its table output,
and its guideline category description. Your task is to rate the
description on one metric. Please make sure you read and understand these
instructions carefully. Please keep this document open while reviewing,
and refer to it as needed.

Evaluation Criteria:
Coherence (1-5) - the collective quality of all sentences. We align this
dimension with the DUC quality question of structure and coherence whereby
 the description should be well-structured and well-organized. The
descripition should not just be a heap of related information, but should
build from sentence to sentence to a coherent body of information about a
topic.

Evaluation Steps:
1. Read the code, its table output, and its guideline category description
 carefully and identify the main topic and key points.
2. Read the description and compare it to the source text including the
code, its table output, and its guideline category description. Check if
the description covers the main topic and key points of the source text,
and if it presents them in a clear and logical order.
3. Assign a score for coherence on a scale of 1 to 5, where 1 is the
lowest and 5 is the highest
based on the Evaluation Criteria.

Example:
Source Text:
{{Guideline Category Description} {Code} {Table}}
Description:
{{Description}}
Evaluation Form (scores ONLY):
- Coherence:
```

```
You will be given one description written for a code, its table output,
and its guideline category description. Your task is to rate the
description on one metric. Please make sure you read and understand these
```

instructions carefully. Please keep this document open while reviewing, and refer to it as needed.

Evaluation Criteria:
Correctness (1-5) - Determine whether the actual output is factually correct based on the expected output.

Evaluation Steps:
1. Read the code, its table output, and its guideline category description carefully and identify the main topic and key points.
2. Read the description and compare it to the source text including the code, its table output, and its guideline category description. Check if the description covers the main topic and key points of the source text, and if it presents them in a clear and logical order.
3. Assign a score for correctness on a scale of 1 to 5, where 1 is the lowest and 5 is the highest
based on the Evaluation Criteria.
Example:
Source Text:
{{Guideline Category Description} {Code} {Table}}
Description:
{{Description}}
Evaluation Form (scores ONLY):
- Correctness:

You will be given one description written for a code, its table output, and its guideline category description. Your task is to rate the description on one metric. Please make sure you read and understand these instructions carefully. Please keep this document open while reviewing, and refer to it as needed.

Evaluation Criteria:
Consistency (1-5) - the generated description is written in the correct guideline category. A factually consistent description contains only statements that are entailed by the source document. Annotators were also asked to penalize descriptions that contained hallucinated facts.

```
Evaluation Steps:
1. Read the code, its table output, and its guideline category description
 carefully and identify the main topic and key points.
2. Read the description and compare it to the source text including the
code, its table output, and its guideline category description. Check if
the description covers the main topic and key points of the source text,
and if it presents them in a clear and logical order.
3. Assign a score for consistency on a scale of 1 to 5, where 1 is the
lowest and 5 is the highest
based on the Evaluation Criteria.
Example:
Source Text:
{{Guideline Category Description} {Code} {Table}}
Description:
{{Description}}
Evaluation Form (scores ONLY):
- Consistency:
```

```
You will be given one description written for a code, its table output,
and its guideline category description. Your task is to rate the
description on one metric. Please make sure you read and understand these
instructions carefully. Please keep this document open while reviewing,
and refer to it as needed.

Evaluation Criteria:
Fluency (1-5) - the quality of individual sentences. Drawing again from
the DUC quality guidelines, sentences in the summary "should have no
formatting problems, capitalization errors or obviously ungrammatical
sentences (e.g., fragments, missing components) that make the text
difficult to read."

Evaluation Steps:
1. Read the code, its table output, and its guideline category description
 carefully and identify the main topic and key points.
2. Read the description and compare it to the source text including the
code, its table output, and its guideline category description. Check if
the description covers the main topic and key points of the source text,
and if it presents them in a clear and logical order.
```

```
3. Assign a score for fluency on a scale of 1 to 5, where 1 is the lowest
and 5 is the highest
based on the Evaluation Criteria.
Example:
Source Text:
{{Guideline Category Description} {Code} {Table}}
Description:
{{Description}}
Evaluation Form (scores ONLY):tab:example4
- Fluency:
```

B.9 Table Preprocess Example

B.9.1 An example of Original Table Crawled from Kaggle Notebooks

```
<div>\n
    <style scoped>
        \n .dataframe tbody tr th:only-of-type {\n vertical-align: middle;\
        n }\n\n .dataframe tbody tr th {\n vertical-align: top;\n }\n\n .
        dataframe thead th {\n text-align: right;\n }\n
    </style>\n
    <table border=\ "1\" class=\ "dataframe\">\n
        <thead>\n
            <tr style=\ "text-align: right;\">\n
                <th></th>\n
                <th>count</th>\n
                <th>mean</th>\n
                <th>std</th>\n
                <th>min</th>\n
                <th>25%</th>\n
                <th>50%</th>\n
                <th>75%</th>\n
                <th>max</th>\n </tr>\n </thead>\n
        <tbody>\n
```

```
<tr>\n
    <th>PassengerId</th>\n
    <td>418.0</td>\n
    <td>1100.500000</td>\n
    <td>120.810458</td>\n
    <td>892.00</td>\n
    <td>996.2500</td>\n
    <td>1100.5000</td>\n
    <td>1204.75</td>\n
    <td>1309.0000</td>\n </tr>\n
<tr>\n
    <th>Pclass</th>\n
    <td>418.0</td>\n
    <td>2.265550</td>\n
    <td>0.841838</td>\n
    <td>1.00</td>\n
    <td>1.0000</td>\n
    <td>3.0000</td>\n
    <td>3.00</td>\n
    <td>3.0000</td>\n </tr>\n
<tr>\n
    <th>Age</th>\n
    <td>332.0</td>\n
    <td>30.272590</td>\n
    <td>14.181209</td>\n
    <td>0.17</td>\n
    <td>21.0000</td>\n
    <td>27.0000</td>\n
    <td>39.00</td>\n
    <td>76.0000</td>\n </tr>\n
<tr>\n
    <th>SibSp</th>\n
    <td>418.0</td>\n
    <td>0.447368</td>\n
    <td>0.896760</td>\n
    <td>0.00</td>\n
    <td>0.0000</td>\n
    <td>0.0000</td>\n
```

```
            <td>1.00</td>\n
            <td>8.0000</td>\n </tr>\n
        <tr>\n
            <th>Parch</th>\n
            <td>418.0</td>\n
            <td>0.392344</td>\n
            <td>0.981429</td>\n
            <td>0.00</td>\n
            <td>0.0000</td>\n
            <td>0.0000</td>\n
            <td>0.00</td>\n
            <td>9.0000</td>\n </tr>\n
        <tr>\n
            <th>Fare</th>\n
            <td>417.0</td>\n
            <td>35.627188</td>\n
            <td>55.907576</td>\n
            <td>0.00</td>\n
            <td>7.8958</td>\n
            <td>14.4542</td>\n
            <td>31.50</td>\n
            <td>512.3292</td>\n </tr>\n </tbody>\n</table>\n</div>"
```

B.9.2 Table preprocessing from the the original table in H.1

After table preprocessing, table is

```
          | count | mean | std | min | 25% | 50% | 75% | max|
PassengerId |418.0 |1100.500000 |120.810458 |892.00 |996.2500 |1100.5000
|1204.75 | 1309.0000|
Pclass |418.0 |2.265550 | 0.841838 |1.00 | 1.0000 |3.0000 | 3.00 |3.0000|
Age |332.0 |30.272590 |14.181209 | 0.17 | 21.0000 | 27.0000 | 39.00
|76.0000|
SibSp |418.0 |0.447368 | 0.896760 | 0.00 | 0.0000 | 0.0000 | 1.00 |
8.0000|
Parch |418.0 |0.392344 | 0.981429 | 0.00 | 0.0000 | 0.0000 | 0.00 |
9.0000|
```

```
Fare |417.0 |35.627188 | 55.907576 | 0.00 | 7.8958 | 14.4542 | 31.50 |
512.3292|
```

B.9.3 Table Curation from the the table in H.2

In the above example, the ground truth description is: “From the count column, we find that some variables have missing values"". Based on the Table Curation method in Section 4.4.2, we extract the “count"" keyword from the ground truth description and extract this related column to generate the final table.

```
           |count |
PassengerId |418.0 |
Pclass |418.0 |
Age |332.0 |
SibSp |418.0 |
Parch |418.0 |
Fare |417.0 |
```

Guideline	N	Description	Example
Value	286 (7.29%)	Get the exact data attribute values for a set of criteria	`The mean survived rate is 38.3 denoting most of the passengers did not survive`
Difference	138 (3.52%)	A comparison between at least two distinct attributes within the target object, or a comparison between the target object and previously measured values.	`The difference though narrows down considerably if we were to consider groups of 2 woman travelers`
Trend	31 (0.79%)	Indicates a general tendency over a period of time.	`Table is displayed in a descending trend in accuracy`
Proportion	120 (3.06%)	Measure the proportion of selected data attribute(s) within a specified set	`8 of 10 passengers have parents`
Categorization	74 (1.89%)	Select the data attribute(s) that meet the condition.	`1 denotes survived while 0 denotes not survived`
Distribution	127 (3.20%)	Show the amount of shared value for the selected data attributes or present a breakdown of all data attributes.	`Fare value range from 7 to 13`
Rank	73 (1.86%)	Sort data attributes by their values and display a breakdown of selected attributes.	`Selecting the top 3 classifiers for model prediction`
Association	165 (4.21%)	Identify the useful relationship between two or more data attributes.	`These two passengers are in the same PClass`
Extreme	227 (5.78%)	Identify the data cases that are the most extreme in relation to the data attributes or within a specific range.	`Model 1 has the highest accuracy`
Outlier	257 (6.55%)	Determine whether there are unexpected data attributes or statistically significant outliers.	`Age column has some missing values`
Aggregation	125 (3.19%)	Calculate the descriptive statistical indicators (e.g., average, sum, count, etc.) based on the data attributes.	`There are 2 classes in the Deck`
Goal	771 (19.64%)	Express user's goal (what value or function they tend to use for later research).	`We use the Gaussian Process Classifier to plot the confusion matrix`
Reason	276 (7.03%)	Express reasons using the data from the table or explain why certain functions are used or why a task is performed.	`We delete the Cabin column, delete 2 rows for Embarked, and since Age plays some role we can ...`
Feature Engineer	393 (10.02%)	The process of selecting, transforming, extracting, combining, and manipulating raw data to generate variables for analysis or predictive modeling.	`Delete Name and Ticket due to its high cardinality`
Complementary Details	870 (22.17%)	Express additional contextual elements and supporting information to enhance understanding of the primary content.	`Column details, counts, and data types provide supplementary technical information about the Titanic passenger dataset structure.`

Table B.7: We identify 15 guideline categories based on the types of descriptions in the Markdown cells below the code whose output is a table.